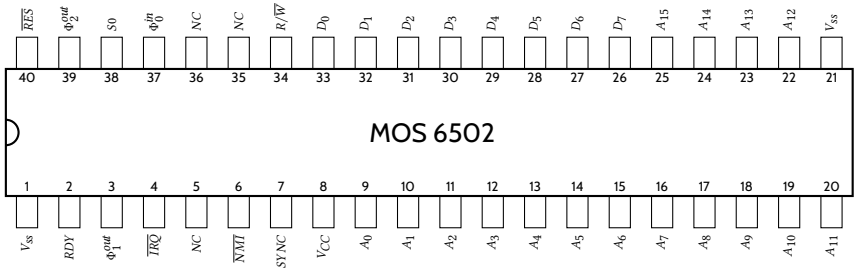
Project Area51
Loose, Maidstone
ME15 OAS, United Kingdom

Volume

Developing for Microcomputers


$\overline{RES}$ Φ_2^{out} $S0$ Φ_0^{in} NC NC $R/\overline{W}$ D_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7 A_{15} A_{14} A_{13} A_{12} V_{ss}
40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21
MOS 6502
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
V_{ss} RDY Φ_1^{out} $\overline{IRQ}$ NC $\overline{NMI}$ $SYNC$ V_{CC} A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10} A_{11}

Peter Mount

Project Area51

Second Edition

Developing for Microcomputers

Developing for Microcomputers

Peter Mount

May 14, 2026

Project Area51

Copyright
This book is copyright 2026 Peter T Mount.


It is made available under the Creative Commons, Attribution, Non-Commercial, Share-Alike licence.

Colophon
This document was typeset with the help of LaTeX using the retepbook class.
The source code of this book is available at:
`https://github.com/peter-mount/developingmicrocomputers`
The LaTeXbook class used is at:
`https://github.com/peter-mount/tex-retepbook`

Publisher
First printed in May 2026 by Project Area51

Always code as if the person who ends up
maintaining your code will be a violent psychopath…

who knows where you live.

– Anon

Contents

Dedication v

Contents vii

Second Edition xlvii

Completion xlviii

Basic 1

1 Introduction to BASIC 3

2 Basic Programming Concepts 5
- 2.1 Program Lines . . . 5
  - 2.1.1 Multiple statements per line . . . 6
- 2.2 Listing a program . . . 6
- 2.3 Running a program . . . 6
- 2.4 Storing and Retrieving a program . . . 7
  - 2.4.1 Retrieving and running a program . . . 7
- 2.5 Variables . . . 8
  - 2.5.1 Numeric Variables . . . 8
  - 2.5.2 Integer Variables . . . 8
  - 2.5.3 String Variables . . . 8
- 2.6 Decisions . . . 9
  - 2.6.1 IF THEN . . . 9
  - 2.6.2 IF THEN ELSE . . . 9
- 2.7 Program Flow . . . 10
  - 2.7.1 Goto . . . 10
  - 2.7.2 Subroutines . . . 10
- 2.8 Inline Data . . . 11

3 BBC Basic 13
- 3.1 Variables . . . 14
  - 3.1.1 Number variables . . . 14
  - 3.1.2 String variables . . . 14
  - 3.1.3 Integer variables . . . 14

3.2 Commands . . . 15
3.2.1 AUTO enter line numbers . . . 15
3.2.2 DELETE lines . . . 16
3.2.3 RENUMBER lines . . . 17
3.3 Statements . . . 18
3.3.1 IF Execute code if condition true . . . 19
3.4 Functions . . . 20
3.4.1 ABS Absolute Value . . . 20
3.4.2 ACS Arc-cosine . . . 20
3.4.3 ADVAL Analogue to digital converter value . . . 21

4 Commodore Basic 23

5 Sinclair Basic 25
5.1 Statements . . . 25
5.1.1 DATA store data . . . 26
5.1.2 FOR perform loop . . . 27
5.1.3 IF Execute code if condition true . . . 28
5.1.4 GO SUB Call Subroutine . . . 29
5.1.5 GO TO line . . . 30
5.1.6 LET Set Variable . . . 31
5.1.7 READ data . . . 32
5.1.8 RESTORE set start of data . . . 33
5.1.9 RETURN from Subroutine . . . 34

6 TI Basic 35
6.1 Variables . . . 35
6.1.1 Numberic variables . . . 35
6.1.2 String variables . . . 36
6.2 Commands . . . 37
6.3 Statements . . . 38
6.3.1 DATA . . . 38
6.3.2 DISPLAY . . . 38
6.3.3 END . . . 39
6.3.4 FOR perform loop . . . 40
6.3.5 GOTO line . . . 41
6.3.6 IF Execute code if condition true . . . 42
6.3.7 LET Set Variable . . . 42

6.3.8 ON-GOTO line . . . 43
6.3.9 NEXT . . . 43
6.3.10 INPUT . . . 44
6.3.11 PRINT . . . 45
6.3.12 READ . . . 48
6.3.13 REMark . . . 48
6.3.14 RESTORE . . . 48
6.3.15 STOP . . . 49
6.4 Functions . . . 50
6.5 Subprograms . . . 51
6.5.1 CALL CHAR . . . 51
6.5.2 CALL CLEAR . . . 51
6.5.3 CALL COLOR . . . 52
6.5.4 CALL GCHAR . . . 53
6.5.5 CALL HCHAR . . . 55
6.5.6 CALL SCREEN . . . 55
6.5.7 CALL SOUND . . . 56
6.5.8 CALL VCHAR . . . 59

Assembly Language **61**

7 Basic Assembly Language Concepts **65**
7.1 Decimal Numbers . . . 65
7.2 Binary Numbers . . . 66
7.3 Hexadecimal Numbers . . . 66
7.4 Octal Numbers . . . 67
7.5 Bytes . . . 68
7.5.1 Nibbles . . . 68
7.6 Multiple Byte Values . . . 69
7.6.1 Endianess . . . 69
7.7 Binary Coded Decimal (BCD) . . . 70
7.8 Signed Numbers . . . 71
7.8.1 Two's complement . . . 71
7.8.2 One's complement . . . 73
7.8.3 Signed bit only . . . 74

8 Flags **75**
8.1 Carry . . . 75
8.2 Zero . . . 76
8.3 Negative . . . 76

9 Program Flow **79**
9.1 Program counter . . . 79
9.2 Jumps . . . 79
9.2.1 Relative Jumps . . . 80
9.2.2 Long Relative Jumps . . . 81
9.2.3 Absolute Jumps . . . 81
9.2.4 Banked Jumps . . . 81
9.2.5 Conditional Branches . . . 82
9.3 Subroutines . . . 84
9.3.1 Calling a Subroutine . . . 84
9.3.2 Recursion . . . 85
9.3.3 Returning from a Subroutine . . . 86
9.3.4 Conditional Subroutine Calls . . . 87

10 Boolean Logic **91**
10.1 Logical And . . . 91
10.2 Logical Or . . . 92
10.3 Logical Exclusive OR . . . 93
10.3.1 Exclusive OR on the 6502 . . . 93
10.3.2 Exclusive OR on the Z80 . . . 93
10.4 Logical Complement . . . 94
10.4.1 Complement on the 6502 . . . 94
10.4.2 Complement on the 65816 . . . 95
10.4.3 Complement on the Z80 . . . 95

11 Shifts and Rotates **97**
11.1 Shift and Rotate Left . . . 98
11.1.1 Rotate Left with Carry . . . 98
11.1.2 Rotate Left with Branch Carry . . . 98
11.1.3 Shift Left . . . 98
11.2 Shift and Rotate Right . . . 99
11.2.1 Rotate Right with Carry . . . 99
11.2.2 Rotate Right with Branch Carry . . . 99

11.2.3 Shift Right . . . 100
11.3 Multiple Byte Shifts . . . 101
11.3.1 Shift multiple left . . . 101
11.3.2 Shift multiple right . . . 102

12 Arithmetic 103
12.1 Increment and Decrement . . . 103
12.1.1 6502 . . . 103
12.1.2 65816 . . . 104
12.1.3 Z80 . . . 104
12.2 Addition . . . 104
12.2.1 Multibyte addition . . . 105
12.2.2 Addition on the 6502 . . . 107
12.2.3 Addition on the 65816 . . . 107
12.2.4 Addition on the Z80 . . . 108
12.3 Subtraction . . . 109
12.3.1 Multibyte subtraction . . . 110
12.3.2 Subtraction on the 6502 . . . 112
12.3.3 Subtraction on the 65816 . . . 112
12.3.4 Subtraction on the Z80 . . . 113
12.4 Comparisons . . . 114
12.4.1 Numeric comparison . . . 114
12.4.2 Binary comparison . . . 114
12.5 Multiplication . . . 115
12.5.1 Multiplication on the 6502 . . . 116
12.6 Division . . . 117

THE 6502 MICROPROCESSOR 119

13 Vectors 121
13.1 Vectors on the 6502 and 65C02 . . . 121
13.1.1 Reset . . . 121
13.1.2 IRQ . . . 121
13.1.3 NMI . . . 122
13.2 Vectors on the 65816 . . . 123
13.2.1 Reset . . . 123
13.2.2 IRQ . . . 123

13.2.3 COP 124
13.2.4 ABORT 124

14 Registers 125
14.1 6502 and 65C02 125
14.1.1 A Accumulator 125
14.1.2 X and Y index registers 125
14.1.3 PC Program Counter 125
14.1.4 S Stack register 126
14.1.5 P Processor Flags 126
14.2 65816 126
14.2.1 P Processor Flags 126
14.2.2 A Accumulator 126
14.2.3 B Accumulator 127
14.2.4 C Accumulator 127
14.2.5 X and Y index registers 127
14.2.6 S Stack register 127
14.2.7 DP Direct Page 127
14.2.8 Data Bank register 127
14.2.9 Program Bank register 128

15 Status Register 129
15.1 Common to all processors 130
15.1.1 C Carry 130
15.1.2 D Decimal Mode 130
15.1.3 I Interrupt Disable 130
15.1.4 N Negative 130
15.1.5 V Overflow 130
15.1.6 Z Zero 130
15.2 Specific to 8-bit processors 131
15.2.1 B Break 131
15.3 Specific to 16-bit processors 131
15.3.1 M Memory and Accumulator mode 131
15.3.2 X Index Register mode 131
15.3.3 E Emulation 131

16 6502 Instruction Set 133
16.1 ADC Add with Carry 134

16.2 AND Bitwise AND A with Memory . . . 136
16.3 ASL Shift Memory or Accumulator Left . . . 137
16.4 BCC Branch if Carry Clear . . . 138
16.5 BCS Branch if Carry Set . . . 139
16.6 BEQ Branch if Equal . . . 140
16.7 BIT Test Memory Bits . . . 141
16.8 BMI Branch if Minus . . . 143
16.9 BNE Branch if Not equal . . . 144
16.10 BPL Branch if Positive . . . 145
16.11 BRA Branch Always . . . 146
16.12 BRK Software Interrupt . . . 147
16.12.1 6502, 65C02 and emulation mode . . . 148
16.12.2 65816 native mode . . . 148
16.12.3 Instruction size discrepancy . . . 149
16.12.4 BBC Micro & Acorn Electron usage . . . 149
16.13 BRL Branch Always Long . . . 150
16.14 BVC Branch if Overflow Clear . . . 151
16.15 BVS Branch if Overflow Set . . . 152
16.16 CLC Clear Carry Flag . . . 153
16.17 CLD Clear Decimal Flag . . . 154
16.18 CLI Enable Hardware Interrupts . . . 155
16.19 CLV Clear Overflow Flag . . . 156
16.20 CMP Compare Accumulator with Memory . . . 157
16.21 COP CO-Processor . . . 158
16.21.1 6502 Emulation Mode . . . 159
16.21.2 65816 Native Mode . . . 159
16.22 CPX Compare X with Memory . . . 160
16.23 CPY Compare Y with Memory . . . 161
16.24 DEC Decrement by one A or Memory . . . 162
16.25 DEX Decrement X by one . . . 163
16.26 DEY Decrement Y by one . . . 164
16.27 EOR Exclusive OR . . . 165
16.27.1 Examples . . . 166
16.28 INC Increment by one A or Memory . . . 167
16.29 INX Increment X by one . . . 168
16.30 INY Increment Y by one . . . 169
16.31 JMP Jump to address . . . 170
16.31.1 JML alternate mnemonic . . . 171

16.31.2 JMP (addr) on the 65C02 . . . 171
16.31.3 JMP (addr) bug on the 6502 . . . 171
16.32 JSR Jump to subroutine . . . 172
16.32.1 JSL alternate mnemonic . . . 173
16.32.2 JSR with 16-bit addresses . . . 173
16.32.3 JSR long . . . 173
16.33 LDA Load Accumulator from Memory . . . 174
16.34 LDX Load X from Memory . . . 175
16.35 LDY Load Y from Memory . . . 176
16.36 LSR Logical Shift Memory or Accumulator Right . . . 177
16.37 MVN Move block Next . . . 178
16.38 MVP Move block Previous . . . 179
16.39 NOP No Operation . . . 180
16.40 ORA OR Accumulator with Memory . . . 181
16.41 PEA Push Effective Absolute Address . . . 182
16.42 PEI Push Effective Indirect Address . . . 183
16.43 PER Push Relative Indirect Address . . . 184
16.44 PHA Push Accumulator . . . 185
16.45 PHB Push Data Bank Register . . . 186
16.46 PHD Push Direct Page Register . . . 187
16.47 PHK Push Direct Page Register . . . 188
16.48 PHP Push Status Flags . . . 189
16.49 PHX Push Index Register X . . . 190
16.50 PHY Push Index Register Y . . . 191
16.51 PLA Pull Accumulator . . . 192
16.52 PLB Pull Data Bank Register . . . 193
16.53 PLD Pull Direct Page Register . . . 194
16.54 PLP Pull Status Flags . . . 195
16.55 PLX Pull Index Register X . . . 197
16.56 PLY Pull Index Register Y . . . 198
16.57 REP Reset Status Bits . . . 199
16.58 ROL Rotate Memory or Accumulator Left . . . 200
16.59 ROR Rotate Memory or Accumulator Right . . . 201
16.60 RTI Return from Interrupt . . . 202
16.61 RTL Return from long subroutine . . . 204
16.62 RTS Return from subroutine . . . 205
16.63 SBC Subtract with Carry . . . 206
16.64 SEC Set Carry . . . 208

16.65 SED Set Decimal Mode . . . 209
16.66 SEI Disable Hardware Interrupts . . . 210
16.67 SEP Set Status Bits . . . 211
16.68 STA Store Accumulator to Memory . . . 212
16.69 STP Stop Processor . . . 213
16.70 STX Store X to Memory . . . 214
16.71 STY Store Y to Memory . . . 215
16.72 STZ Store Zero to Memory . . . 216
16.73 TAX Transfer Accumulator to X . . . 217
16.73.1 Transfer size . . . 218
16.74 TAY Transfer Accumulator to Y . . . 219
16.75 TCD Transfer 16-Bit Accumulator to Direct Page Register . . . 220
16.76 TCS Transfer Accumulator to Stack Pointer . . . 221
16.77 TDC Transfer Direct Page Register to 16-Bit Accumulator . . . 222
16.78 TRB Test and Reset Memory Bits against Accumulator . . . 223
16.78.1 Size of memory affected . . . 224
16.79 TSB Test and Set Memory Bits against Accumulator . . . 225
16.80 TSC Transfer Stack Pointer to 16-Bit Accumulator . . . 226
16.81 TSX Transfer Stack Pointer to X . . . 228
16.82 TXA Transfer X to Accumulator . . . 229
16.83 TXS Transfer X to Stack Pointer . . . 230
16.84 TXY Transfer X to Y . . . 231
16.85 TYA Transfer Y to Accumulator . . . 232
16.86 TYX Transfer Y to X . . . 233
16.87 WAI Wait until hardware Interrupt . . . 234
16.88 WDM Reserved . . . 235
16.89 XBA Exchange the B and A Accumulators . . . 236
16.90 XCE Exchange Carry and Emulation Bits . . . 237
16.90.1 Switching to 6502 emulation mode . . . 238
16.90.2 Switching to native mode . . . 238

THE Z80 MICROPROCESSOR 239

17 Registers 241
17.1 Accumulator and Flag registers . . . 242
17.1.1 Using the Stack . . . 242
17.1.2 Selecting the alternate pair . . . 242

17.2 General Purpose Registers 243
17.2.1 Using the Stack 243
17.2.2 B, C and BC Registers 244
17.2.3 D, E and DE Registers 245
17.2.4 H, L and HL Registers 245
17.3 Special Purpose Registers 245
17.4 PC Register 245
17.5 SP Register 246
17.6 IX and IY Registers 246
17.6.1 Using the Stack 246
17.7 I Interrupt Page Address Register 247
17.8 R Memory Refresh Register 247

18 Flags **249**
18.1 C Carry Flag 249
18.1.1 Rotations and the Carry Flag 250
18.1.2 Shifts and the Carry Flag 250
18.1.3 Setting the Carry Flag 250
18.1.4 Clearing the Carry Flag 251
18.2 Z Zero Flag 251
18.3 S Sign Flag 252
18.4 P/V Parity/Overflow flag 252
18.4.1 Parity 252
18.4.2 Arithmetic Overflow 253
18.5 Lesser used flags 254
18.5.1 H Half Carry Flag 254
18.5.2 N Add/Subtract Flag 254

19 Arithmetic and Logic **255**
19.1 Increment and Decrement 255
19.2 Negation and Complement 255
19.3 Addition and Subtraction 256
19.3.1 With Carry or Borrow 256
19.3.2 Without Carry or Borrow 256
19.4 Compare 257
19.4.1 Branching on results 257
19.5 Logic 260

19.6 Rotates and Shifts . . . 261
19.6.1 Shift and Rotate Left . . . 261
19.6.2 Shift and Rotate Right . . . 262
19.7 Bit Manipulation . . . 263
19.8 Binary Coded Decimal . . . 264
19.8.1 BCD Adjustment . . . 264
19.8.2 BCD Rotate Left and Right . . . 264

20 Input/Output 265
20.1 Input from a device . . . 266
20.1.1 Reading from Port Example . . . 266
20.2 Output to a device . . . 268
20.3 The number of ports problem . . . 268

21 Block instructions 271
21.1 Block Load . . . 271
21.2 Compare Memory . . . 272
21.2.1 Compare from start of block . . . 272
21.2.2 Compare from end of block . . . 272
21.3 Block Input . . . 273
21.3.1 Address Bus . . . 273
21.4 Block Output . . . 274
21.4.1 Address Bus . . . 274

22 Z80 Instruction Set 275
22.1 ADC Add with Carry . . . 276
22.2 ADD without Carry . . . 278
22.3 AND Logical AND . . . 280
22.4 BIT Test bit value . . . 281
22.5 Call Subroutine . . . 283
22.6 CCF Complement Carry Flag . . . 285
22.7 CP Compare to Accumulator . . . 286
22.8 CPD Compare with decrement . . . 288
22.8.1 Compare Block Instructions . . . 289
22.9 CPDR Block compare with decrement . . . 290
22.10 CPI Compare with increment . . . 291
22.11 CPIR Block compare with increment . . . 292
22.12 CPL Complement Accumulator . . . 293

22.13 DAA Decimal Adjust Accumulator 294
22.14 DEC Decrement register or memory 296
22.15 DI Disable Maskable Interrupts 298
22.16 DJNZ Decrement B and jump on non-zero 299
22.17 EI Enable Maskable Interrupts 301
22.18 EX Exchange Registers 302
22.19 EXX Exchange Alternate Registers 304
22.20 HALT the CPU 305
22.21 IM Set Interrupt Mode 306
22.21.1 Interrupt Mode 0 307
22.21.2 Interrupt Mode 1 307
22.21.3 Interrupt Mode 2 307
22.22 IN Load register from Port 308
22.23 INC Increment register or memory 310
22.24 IND Input with Decrement 312
22.24.1 Input Block Instructions 313
22.25 INDR Block input with Decrement 314
22.26 INI Input with Increment 315
22.27 INIR Block input with Increment 316
22.28 JP Jump to location 317
22.29 JR Jump to relative location 319
22.30 LD Load register or memory 320
22.31 LDD Block load with Decrement 330
22.31.1 Block Load Instructions 331
22.32 LDDR Block load with Decrement 332
22.33 LDI Block Load with Increment 333
22.34 LDIR Block load with Increment 334
22.35 NEG Negate Accumulator 335
22.36 NOP No Operation 336
22.37 OR Bit or accomulator and operand 337
22.38 OTDR Block output with Decrement 339
22.39 OTIR Block output with Increment 340
22.40 OUT Output register to Port 341
22.41 OUTD Block output with Decrement 343
22.41.1 Output Block Instructions 344
22.42 OUTI Block output with Increment 345
22.43 POP register pair from stack 346
22.44 PUSH register pair onto stack 348

22.45 RES Reset bit b in operand 350
22.46 RET Return from subroutine 354
22.47 RETI Return from interrupt 355
22.48 RETN Return from non-maskable interrupt 356
22.49 RL Rotate left through carry 357
22.50 RLA Rotate Accumulator left through carry 359
22.51 RLC Rotate left with branch carry 360
22.52 RLCA Rotate accumulator left with branch carry 362
22.53 RLD 363
22.54 RR Rotate right through carry 364
22.55 RRA Rotate Accumulator right through carry 366
22.56 RRCA Rotate accumulator right with branch carry 367
22.57 RRC Rotate right with branch carry 368
22.58 RRD 370
22.59 RST Restart at p 371
22.59.1 RST naming 372
22.59.2 RST `0x00` 373
22.59.3 RST `0x38` 373
22.60 SBC Subtract with borrow 375
22.61 SCF Set carry flag 377
22.62 SET bit b in operand 378
22.63 SLA Arithmetic shift left 382
22.64 SRA Arithmetic shift right 384
22.65 SRL Arithmetic shift right 386
22.66 SUB Subtract without carry 388
22.67 XOR Exclusive or 390

Assemblers **393**

23 BBC Basic Assembler **397**
23.1 Address modes 398
23.1.1 Implicit 398
23.1.2 Absolute address 398
23.1.3 zero-page address 399
23.1.4 Indirect 399
23.1.5 Indexed 399
23.1.6 zero-page Indexed 400

23.1.7 Relative . . . 400
23.1.8 Accumulator . . . 400
23.2 Assembly Destination . . . 401
23.3 Multiple Passes . . . 403
23.4 Directives . . . 405
23.4.1 [ ] . . . 405
23.4.2 EQUB . . . 405
23.4.3 EQUW . . . 405
23.4.4 EQUD . . . 406
23.4.5 EQUS . . . 406
23.4.6 OPT . . . 406

24 BeebASM 409
24.1 Installing BeebASM . . . 410
24.1.1 Compilation . . . 410
24.1.2 Installation . . . 410
24.2 Command line . . . 411
24.3 Source file syntax . . . 412
24.3.1 Comments . . . 412
24.3.2 Instructions . . . 412
24.3.3 Labels . . . 413
24.3.4 Local labels . . . 413
24.4 Variables . . . 416
24.4.1 Conditional assignment . . . 416
24.5 Operators . . . 417
24.5.1 16-bit to 8-bit conversions . . . 418
24.5.2 Numeric Literals . . . 420
24.5.3 String Literals . . . 420
24.6 Directives . . . 421
24.6.1 ALIGN . . . 421
24.6.2 ASM . . . 421
24.6.3 ASSERT . . . 421
24.6.4 CLEAR . . . 422
24.6.5 COPYBLOCK . . . 422
24.6.6 CPU . . . 422
24.6.7 EQUB . . . 423
24.6.8 EQUD . . . 423
24.6.9 EQUS . . . 423

24.6.10 EQUW . . . 423
24.6.11 ERROR . . . 424
24.6.12 FOR . . . 424
24.6.13 GUARD . . . 426
24.6.14 IF . . . 427
24.6.15 INCBIN . . . 429
24.6.16 INCLUDE . . . 429
24.6.17 MACRO . . . 429
24.6.18 MAPCHAR . . . 432
24.6.19 ORG . . . 433
24.6.20 PRINT . . . 433
24.6.21 PUTBASIC . . . 434
24.6.22 PUTFILE . . . 434
24.6.23 PUTTEXT . . . 435
24.6.24 RANDOMIZE . . . 435
24.6.25 SAVE . . . 436
24.6.26 SKIP . . . 437
24.6.27 SKIPTO . . . 437
24.6.28 SOURCELINE . . . 437
24.7 Contants . . . 438
24.7.1 * or P% . . . 438
24.7.2 CPU . . . 438
24.7.3 PI . . . 438
24.7.4 FALSE . . . 438
24.7.5 TRUE . . . 438
24.8 Functions . . . 439
24.8.1 ABS . . . 439
24.8.2 ACS . . . 439
24.8.3 ASC . . . 439
24.8.4 ASN . . . 439
24.8.5 ATN . . . 439
24.8.6 CHR$ . . . 440
24.8.7 COS . . . 440
24.8.8 DEG . . . 440
24.8.9 EXP . . . 440
24.8.10 EVAL . . . 440
24.8.11 INT . . . 441
24.8.12 LEFT$ . . . 441

24.8.13 LEN . . . 441
24.8.14 LOWER$ . . . 441
24.8.15 LN . . . 441
24.8.16 LOG . . . 442
24.8.17 MID$ . . . 442
24.8.18 NOT . . . 442
24.8.19 RAD . . . 442
24.8.20 RIGHT$ . . . 443
24.8.21 RND . . . 443
24.8.22 SIN . . . 443
24.8.23 SGN . . . 443
24.8.24 SQR . . . 444
24.8.25 STR$ . . . 444
24.8.26 STR$~ . . . 444
24.8.27 STRING$ . . . 444
24.8.28 TIME$ . . . 445
24.8.29 TAN . . . 445
24.8.30 UPPER$ . . . 445
24.8.31 VAL . . . 445

25 Lancaster Assembler 447
25.1 Using the 65C02 assembler Version 1.60 . . . 447
25.2 Introduction . . . 448
25.2.1 Installing the Assembler . . . 448
25.3 The Assembler's CLI mode . . . 449
25.4 Changing screen mode . . . 449
25.5 Creating source files . . . 450
25.5.1 Using WordWise . . . 451
25.5.2 Using View . . . 451
25.6 Starting an assembly . . . 452
25.6.1 Assembly on a second processor . . . 457
25.6.2 The object code buffer . . . 458
25.6.3 Returning to the calling language . . . 458
25.6.4 Use of control keys during assembly . . . 459
25.7 Source Program Format . . . 461
25.7.1 Syntax of the line . . . 462
25.7.2 The syntax of expressions . . . 464

25.8 The Output Listing 469
25.8.1 The source listing level 469
25.8.2 Source listing format 470
25.8.3 Listing object code bytes 471
25.8.4 Symbol table listing format 472
25.8.5 Error reports 473
25.8.6 Output page control 473
25.8.7 Controlling the output destination 476
25.9 Directives Defining Data and Constants 479
25.9.1 Defining constants 479
25.9.2 Byte and word directives 479
25.9.3 Character string directives 483
25.9.4 The HEX directive 485
25.9.5 The CODE directive 486
25.10 Object File Control Directives 487
25.10.1 Defining the current address 487
25.10.2 Reading the current address 488
25.10.3 Using dummy sections 489
25.10.4 The DS directive 490
25.10.5 Defining object load and execution addresses 491
25.11 Source File Control Directives 493
25.11.1 Chaining source files 493
25.11.2 Including one file into another 494
25.12 Conditional Assembly 495
25.12.1 Assembly conditional on expressions 495
25.12.2 Assembly conditional on the existence of symbols . . 497
25.12.3 Nesting conditionals 497
25.12.4 Listing conditionals 497
25.13 Progress Reporting 499
25.13.1 The DISP, DISP1 and DISP2 directives 499
25.13.2 The WAIT, WAIT1 and WAIT2 directives 500
25.13.3 The QUERY directive 500
25.13.4 The STOP directive 502
25.14 Writing Simple Macros 503
25.14.1 Using macro parameters 503
25.14.2 Specifying macro parameters 504
25.14.3 Nesting macros 505
25.14.4 Redefining opcodes and directives 505

25.14.5 Labels within macros 506
25.15 The Macro Programming Language 509
25.15.1 Sequence Symbols 509
25.15.2 Assembly Time Variables 510
25.15.3 Writing Complex Macros 517
25.15.4 System ATVs 525
25.16 Supported Processors 526
25.16.1 Supported Opcodes 526
25.16.2 Address modes 528
25.17 Directives 529
25.17.1 AGO 529
25.17.2 AIF 529
25.17.3 AIFDEF 529
25.17.4 AIFNDEF 530
25.17.5 ALEN 530
25.17.6 ASC 530
25.17.7 ASET 530
25.17.8 ASTR 531
25.17.9 ASTRIP 531
25.17.10 CASC 531
25.17.11 CHN 532
25.17.12 CHR 532
25.17.13 CLST 533
25.17.14 CODE 533
25.17.15 CSTR 534
25.17.16 DB 534
25.17.17 DDB 534
25.17.18 DEFPARS 535
25.17.19 DEND 535
25.17.20 DFB 535
25.17.21 DFDB 536
25.17.22 DFS 536
25.17.23 DFW 536
25.17.24 DISP 536
25.17.25 DISP1 536
25.17.26 DISP2 537
25.17.27 DO 537
25.17.28 DS 537

25.17.29 DSECT 538
25.17.30 DW 538
25.17.31 ELSE 539
25.17.32 ENDM 539
25.17.33 EQU 539
25.17.34 EXEC 540
25.17.35 FI 540
25.17.36 FIN 540
25.17.37 HEX 541
25.17.38 IF 541
25.17.39 IFDEF 541
25.17.40 IFNDEF 542
25.17.41 INCLUDE 542
25.17.42 INFO 542
25.17.43 LFCOND 543
25.17.44 LOAD 543
25.17.45 LST 544
25.17.46 MACRO 545
25.17.47 MEXIT 545
25.17.48 MLEN 545
25.17.49 MSET 546
25.17.50 MSTR 546
25.17.51 MSTRIP 546
25.17.52 MSW 546
25.17.53 ORG 547
25.17.54 PAGE 547
25.17.55 QUERY 548
25.17.56 REP 548
25.17.57 SFCOND 548
25.17.58 SKP 549
25.17.59 STOP 549
25.17.60 STR 549
25.17.61 SYSCLI 550
25.17.62 SYSVDU 551
25.17.63 SYSVDU1 551
25.17.64 SYSVDU2 551
25.17.65 TABS 552
25.17.66 TTL 552

25.17.67 TIME . . . 553
25.17.68 WAIT . . . 553
25.17.69 WAIT1 . . . 553
25.17.70 WAIT2 . . . 553
25.18 Differences from the ADE Assembler . . . 554
25.19 Differences from previous releases . . . 555
25.20 Customising the assembler . . . 557
25.20.1 The patch area . . . 557
25.20.2 The default TAB position list . . . 558
25.20.3 Source character translation . . . 558
25.20.4 Real time clock handling . . . 559
25.20.5 Character constant and string options . . . 559
25.20.6 Page throws at the end of a listing . . . 559

The BBC Microcomputer 561

BBC Model B 563

26 Memory Map 565

BBC Master 128 567

27 Memory Map 569

28 Configuration 571
28.1 Default configuration . . . 572
28.2 ANFS . . . 573
28.2.1 *CONFIGURE FS <x.xxx> or <Name> . . . 573
28.2.2 *CONFIGURE PS <x.xxx> or <Name> . . . 573
28.2.3 *CONFIGURE SPACE/NOSPACE . . . 574
28.3 *CONFIGURE Baud <0...8> . . . 575
28.4 *CONFIGURE Boot / NoBoot . . . 575
28.5 *CONFIGURE Caps / NoCaps . . . 576
28.6 *CONFIGURE Data <0...7> . . . 576
28.7 *CONFIGURE Delay <0...255> . . . 577
28.8 *CONFIGURE Dir/NoDir . . . 577

28.9 *CONFIGURE FDrive <0...7> 577
28.10 *CONFIGURE File <0...15> 578
28.11 *CONFIGURE Hard/Floppy 578
28.12 *CONFIGURE Ignore <0...255> 578
28.13 *CONFIGURE Lang <0...15> 579
28.14 *CONFIGURE Mode <0...7, 128...135> 579
28.15 *CONFIGURE Print <0...4> 579
28.16 *CONFIGURE Quiet/Loud 580
28.17 *CONFIGURE Repeat <0...255> 580
28.18 *CONFIGURE Scroll / NoScroll 580
28.19 *CONFIGURE Tube / NoTube 581
28.20 *CONFIGURE View Setup <xxx> 581

Acorn Electron **583**

MOS
Machine Operating System **585**

29 Keyboards **587**
29.1 Background processing 587
29.2 Break key 588
29.3 Reading keys 589
29.3.1 ASCII 589
29.3.2 INKEY 589
29.3.3 Internal Key Code 590
29.4 Keys and layouts 590
29.4.1 BBC Model B keyboard 590
29.4.2 BBC Master 128 and BBC Master Compact keyboards 590
29.4.3 Acorn Electron keyboard 591

30 Operating System Calls **595**
30.1 OSWRCH - Write character 596
30.2 OSNEWL - Write newline 596
30.3 OSASCI - Write character, call OSNEWL if A=0x0D 597
30.4 OSRDCH - Read character 598
30.5 OSBGET Read byte from open file 599

30.6 OSBPUT Write byte to open file 599
30.7 OSCLI - Pass string to CLI 600
30.8 OSEVEN - Generate an event 602
30.9 OSRDSC - Read byte from Screen or paged ROM 602
30.10 OSBYTE - OS functions by A 603
30.10.1 Parameters 603
30.10.2 Calling OSBYTE 606
30.10.3 Errata 607
30.10.4 0x00 (0) Get OS Version 608
30.10.5 0x01 (1) Write user flag 609
30.10.6 0x02 (2) Specify input stream 610
30.10.7 0x03 (3) Specify output stream 611
30.10.8 0x04 (4) Enable/disable cursor editing 612
30.10.9 0x05 (5) Select printer destination 613
30.10.10 0x06 (6) Write printer ignore character 613
30.10.11 0x07 (7) Set RS-423 receive rate 614
30.10.12 0x08 (8) Set RS-423 transmit rate 614
30.10.13 0x09 (9) Set duration of first colour 615
30.10.14 0x0A (10) Write duration of second colour 615
30.10.15 0x0B (11) Set keyboard auto-repeat delay 616
30.10.16 0x0C (12) Set keyboard auto-repeat rate 616
30.10.17 0x0D (13) Disable event 617
30.10.18 0x0E (14) Enable event 617
30.10.19 0x0F (15) Flush buffers 618
30.10.20 0x10 (16) Select ADC channel 618
30.10.21 0x11 (17) Force ADC conversion 619
30.10.22 0x12 (18) Reset soft function keys 619
30.10.23 0x13 (19) Wait for vertical sync 620
30.10.24 0x14 (20) Explode character definition RAM 620
30.10.25 0x14 (20) Reset character definitions 621
30.10.26 0x15 (21) Flush specific buffer 622
30.10.27 0x16 (22) Increment polling semaphore 623
30.10.28 0x17 (23) Decrement polling semaphore 623
30.10.29 0x18 (24) Select external Sound 624
30.10.30 0x19 (25) Restore default font definitions 624
30.10.31 0x44 (68) Test for sideways RAM 625
30.10.32 0x45 (69) Sideways RAM allocation 626
30.10.33 0x6B (107) Select 1MHz bus/cartridge 627

30.10.34 0x6C (108) Select screen memory for direct access . . 628
30.10.35 0x6D (109) Make temporary file system permanent . . 628
30.10.36 0x70 (112) Select memory for VDU 629
30.10.37 0x71 (113) Select memory for display 630
30.10.38 0x72 (114) Write shadow memory use 630
30.10.39 0x73 (115) Blank or restore palette 631
30.10.40 0x74 (116) Reset sound system 631
30.10.41 0x75 (117) Read VDU status 632
30.10.42 0x76 (118) Make LEDs reflect keyboard status 633
30.10.43 0x77 (119) Close *SPOOL/*EXEC files 633
30.10.44 0x78 (120) Write current keys pressed 634
30.10.45 0x79 (121) Perform keyboard scan 634
30.10.46 0x7A (122) Perform keyboard scan from 0x10 635
30.10.47 0x7B (123) Printer driver going dormant 636
30.10.48 0x7C (124) Clear ESCAPE condition 636
30.10.49 0x7D (125) Set ESCAPE condition 637
30.10.50 0x7E (126) Acknowledge ESCAPE, clear + effects . . . 637
30.10.51 0x7F (127) Check for EOF 638
30.10.52 0x80 (128) Read ADC channel/buffer status 638
30.10.53 0x81 (129) Read key with time limit 641
30.10.54 0x81 (129) Scan for specific key 642
30.10.55 0x81 (129) Read machine type 643
30.10.56 0x82 (130) Read machine high order address 644
30.10.57 0x83 (131) Read top of OS RAM 644
30.10.58 0x84 (132) Read top of user RAM 645
30.10.59 0x85 (133) Read HIMEM for given mode 645
30.10.60 0x86 (134) Read input cursor pos 646
30.10.61 0x87 (135) Read char at text cursor and screen mode . 646
30.10.62 0x88 (136) *CODE 647
30.10.63 0x89 (137) *MOTOR 647
30.10.64 0x8A (138) Insert into buffer 648
30.10.65 0x8B (139) *OPT 649
30.10.66 0x8C (140) *TAPE 650
30.10.67 0x8D (141) *ROM 650
30.10.68 0x8E (142) Enter language ROM 651
30.10.69 0x8F (143) Issue paged ROM service request 651
30.10.70 0x90 (144) *TV 652
30.10.71 0x90 (144) *TV 652

30.10.72 0x91 (145) Get character from buffer 653
30.10.73 0x92 (146) Read from FRED 654
30.10.74 0x93 (147) Write to FRED 655
30.10.75 0x94 (148) Read from JIM 655
30.10.76 0x95 (149) Write to JIM 655
30.10.77 0x96 (150) Read from SHIELA 656
30.10.78 0x97 (151) Write to SHIELA 656
30.10.79 0x98 (152) Examine buffer status 657
30.10.80 0x98 (152) Examine buffer status 658
30.10.81 0x99 (153) Insert character into input buffer 660
30.10.82 0x9A (154) Write to video ULA 661
30.10.83 0x9A (154) Reset flash cycle 661
30.10.84 0x9B (155) Write to video ULA palette register 662
30.10.85 0x9B (155) No operation 662
30.10.86 0x9C (156) Read/Write 6850 control register 663
30.10.87 0x9D (157) Fast Tube BPUT 665
30.10.88 0x9E (158) Read from speech processor 665
30.10.89 0x9E (158) Reserved 666
30.10.90 0x9F (159) Write to speech processor 666
30.10.91 0x9F (159) Reserved 667
30.10.92 0xA0 (160) Read VDU variable 668
30.10.93 0xA1 (161) Read CMOS RAM 669
30.10.94 0xA1 (161) Read EEPROM 669
30.10.95 0xA1 (161) Query EEPROM size 670
30.10.96 0xA2 (162) Write CMOS RAM/EEPROM 670
30.10.97 0xA3 (163) Enable or Disable printer and ADC 671
30.10.98 0xA3 (163) Reserved for application software 671
30.10.99 0xA4 (164) Check for 6502 code 672
30.10.100 0xA5 (165) Read output cursor position 673
30.10.101 0xA6 (166) Base of MOS variables 673
30.10.102 0xA8 (168) ROM Pointer table 674
30.10.103 0xAA (170) ROM Information table 675
30.10.104 0xAC (172) Keyboard Translation Table 676
30.10.105 0xAE (174) VDU Variables Table 676
30.10.106 0xB0 (176) CFS timeout counter 677
30.10.107 0xB1 (177) Input source 677
30.10.108 0xB2 (178) Keyboard semaphore 678
30.10.109 0xB3 (179) Default OSHWM 679

30.10.110 0xB3 (179) Paged ROM polling semaphore 680
30.10.111 0xB4 (180) Current OSHWM 681
30.10.112 0xB5 (181) RS-423 mode 682
30.10.113 0xB6 (182) Character Explosion State 683
30.10.114 0xB6 (182) Printer Ignore Character 683
30.10.115 0xB7 (183) TAPE or ROM Switch 684
30.10.116 0xB8 (184) Video ULA control register 684
30.10.117 0xB9 (185) Video ULA palette register 685
30.10.118 0xB9 (185) Paged ROM polling semaphore 686
30.10.119 0xBA (186) ROM active on last BRK 687
30.10.120 0xBB (187) BASIC ROM number 688
30.10.121 0xBC (188) Current ADC channel being sampled . . . 689
30.10.122 0xBD (189) Maximum ADC channel number 689
30.10.123 0xBE (190) ADC conversion type 690
30.10.124 0xBE (190) ADC conversion type 691
30.10.125 0xBF (191) RS-423 Use Flag 693
30.10.126 0xC0 (192) RS-423 6840 control register 694
30.10.127 0xC0 (192) RS-423 6840 control register 694
30.10.128 0xC1 (193) Read Flash Counter 695
30.10.129 0xC2 (194) Flash Mark Colour Duration 696
30.10.130 0xC3 (195) Flash Space Colour Duration 696
30.10.131 0xC4 (196) Keyboard Auto-repeat delay 697
30.10.132 0xC5 (197) Keyboard Auto-repeat rate 697
30.10.133 0xC6 (198) *EXEC File Handle 698
30.10.134 0xC7 (199) *SPOOL File Handle 698
30.10.135 0xC8 (200) Escape and Break Effect 699
30.10.136 0xC9 (201) Keyboard Disable 700
30.10.137 0xCA (202) Keyboard Status 701
30.10.138 0xCB (203) RS-423 Handshake Level 702
30.10.139 0xCB (203) Electron ULA IRQ mask 703
30.10.140 0xCC (204) RS-423 Suppression Flag 703
30.10.141 0xCC (204) Firm Key Pointer 704
30.10.142 0xCD (205) RS-423 Selection Flag 705
30.10.143 0xCD (205) Firm Key string length 706
30.10.144 0xCE (206) Econet OS call interception status 706
30.10.145 0xCF (207) Econet read char interception status 707
30.10.146 0xD0 (208) Econet write char interception status . . . 707
30.10.147 0xD1 (209) Speech suppression status 708

30.10.148 0xD2 (210) Sound suppression status 708
30.10.149 0xD3 (211) Sound BELL channel 709
30.10.150 0xD4 (212) Sound BELL amplitude/envelope 709
30.10.151 0xD5 (213) Sound BELL pitch 710
30.10.152 0xD6 (214) Sound BELL duration 710
30.10.153 0xD7 (215) Startup message and Boot Options 711
30.10.154 0xD8 (216) Soft Key Length 712
30.10.155 0xD9 (217) Lines printed since last page pause 712
30.10.156 0xDA (218) Number items in VDU queue 713
30.10.157 0xDB (219) TAB character 714
30.10.158 0xDB (219) External sound flag 715
30.10.159 0xDC (220) ESCape character 715
30.10.160 0xDD (221) Input buffer interp Cx 716
30.10.161 0xDE (222) Input buffer interp Dx 717
30.10.162 0xDF (223) Input buffer interp Ex 717
30.10.163 0xE0 (224) Input buffer interp Fx 718
30.10.164 0xE1 (225) Function Key status 718
30.10.165 0xE2 (226) Function Key + SHIFT status 719
30.10.166 0xE2 (226) Function Key + A...P status 720
30.10.167 0xE3 (227) Function Key + CTRL status 721
30.10.168 0xE3 (227) Function Key + Q...Z status 722
30.10.169 0xE4 (228) Function Key + CTRL SHIFT status 723
30.10.170 0xE5 (229) Escape action flag 724
30.10.171 0xE6 (230) Escape effects flag 725
30.10.172 0xE7 (231) User VIA IRQ bit mask 726
30.10.173 0xE7 (231) Reserved for hardware expansion 726
30.10.174 0xE8 (232) RS-423 6850 IRQ bit mask 727
30.10.175 0xE8 (232) Sound semaphore 727
30.10.176 0xE9 (233) System VIA IRQ bit mask 728
30.10.177 0xE9 (233) Soft key pointer 729
30.10.178 0xEA (234) Tube present flag 730
30.10.179 0xEB (235) Speech System present flag 730
30.10.180 0xEC (236) Output stream destination 731
30.10.181 0xED (237) Cursor editing status 731
30.10.182 0xEE (238) Numeric Keypad base 732
30.10.183 0xEF (239) Shadow memory use 733
30.10.184 0xF0 (240) Country Code 734
30.10.185 0xF1 (241) User Flag 735

30.10.186 0xF2 (242) Serial ULA control register 736
30.10.187 0xF2 (242) Last value written to ULA 737
30.10.188 0xF3 (243) Time Clock Switch 738
30.10.189 0xF4 (244) Soft key consistency flag 739
30.10.190 0xF5 (245) Printer destination 740
30.10.191 0xF6 (246) Printer ignore character 740
30.10.192 0xF7 (247) Break Intercept 741
30.10.193 0xFA (250) Select memory for VDU 742
30.10.194 0xFB (251) Select memory for display 742
30.10.195 0xFC (252) Current language ROM 743
30.10.196 0xFD (253) Last reset type 744
30.10.197 0xFE (254) RAM Size 745
30.10.198 0xFE (254) Reserved 745
30.10.199 0xFE (254) Keypad Shift Key Flag 746
30.10.200 0xFF (255) Startup Options 747
30.11 OSWORD - OS functions with a parameter block 751
30.11.1 Parameters 751
30.11.2 MOS OSWORD calls 752
30.11.3 Application and Filesystem calls 752
30.11.4 Unknown OSWORD calls 752
30.11.5 0x00 (0) Read line from current input stream 753
30.11.6 0x01 (1) Read system clock 755
30.11.7 0x02 (2) Write system clock 755
30.11.8 0x03 (3) Read interval timer 756
30.11.9 0x04 (4) Write interval timer 756
30.11.10 0x05 (5) Read byte of I/O processor memory 757
30.11.11 0x06 (6) Write byte of I/O processor memory 758
30.11.12 0x07 (7) Perform a SOUND command 759
30.11.13 0x08 (8) Define an ENVELOPE 761
30.11.14 0x09 (9) Read pixel value 764
30.11.15 0x0A (10) Read character definition 765
30.11.16 0x0B (11) Read palette for given logical colour 766
30.11.17 0x0C (12) Write palette for given logical colour 766
30.11.18 0x0D (13) Read previous and current graphics cursor position 767
30.11.19 0x0E (14) Read real time clock 768
30.11.20 0x0E (14) Read real time clock 770
30.11.21 0x0F (15) Write real time clock 772

30.11.22 0xFF (255) Read/Write I/O processor memory Z80 . . 773

31 Operating System Commands 777
31.1 *CODE x,y . . . 777
31.2 *FX a,x,y Perform OSBYTE call . . . 777
31.3 *LINE text . . . 778
31.4 *MOTOR n . . . 778
31.5 *TAPE n . . . 778
31.6 *TV s,i . . . 779
31.7 Media . . . 780

FILESYSTEMS 781

32 Commands common to all filesystems 783
32.0.1 *CAT . . . 784
32.0.2 *EXEC . . . 784
32.0.3 *LOAD . . . 784
32.0.4 *OPT . . . 785
32.0.5 *RUN . . . 786
32.0.6 *SAVE . . . 786
32.0.7 *SPOOL . . . 786

33 Cassette Filesystem 787

34 ROM Filesystem 789

35 Acorn DFS 791
35.1 Commands . . . 791
35.1.1 *ACCESS . . . 791
35.1.2 *BACKUP . . . 792
35.1.3 *BUILD . . . 792
35.1.4 *CAT . . . 793
35.1.5 *COMPACT . . . 793
35.1.6 *COPY . . . 794
35.1.7 *DELETE . . . 794
35.1.8 *DESTROY . . . 795
35.1.9 *DIR . . . 795
35.1.10 *DRIVE . . . 795

35.1.11 *DUMP 796
35.1.12 *ENABLE 796
35.1.13 *EXEC 796
35.1.14 *INFO 796
35.1.15 *LIB 797
35.1.16 *LIST 797
35.1.17 *LOAD 797
35.1.18 *OPT 1 798
35.1.19 *OPT 4 798
35.1.20 *RENAME 799
35.1.21 *RUN 799
35.1.22 *SAVE 799
35.1.23 *SPOOL 799
35.2 DFS Internals 800
35.2.1 Sector ID's 800
35.2.2 Catalog 800
35.2.3 Identifying and validating a DFS disc image 807

36 Acorn ADFS 811

37 LanManager File System 813
37.1 About the SPROW 10/100 Network interface 813
37.2 Networking Commands 814
37.2.1 *CONFIGURE EMLINK 814
37.2.2 *CONFIGURE EMADVERTISE 815
37.2.3 *CONFIGURE EMADDR 816
37.2.4 *CONFIGURE EMMASK 816
37.2.5 *CONFIGURE GATEWAY 816
37.2.6 *CONFIGURE DNS 816
37.2.7 *EMINFO 817
37.2.8 *EMTEST 817
37.2.9 *PING 818
37.2.10 *STATUS EMLINK 818
37.2.11 *STATUS EMADVERTISE 818
37.2.12 *STATUS EMADDR 818
37.2.13 *STATUS EMMASK 819
37.2.14 *STATUS GATEWAY 819
37.2.15 *STATUS DNS 819

37.3 CIFS Commands . . . 820
37.3.1 *ACCESS . . . 820
37.3.2 *BYE . . . 820
37.3.3 CDIR . . . 820
37.3.4 *COPY . . . 820
37.3.5 *DESTROY . . . 821
37.3.6 *DISMOUNT . . . 821
37.3.7 *DRIVE . . . 821
37.3.8 *DIR . . . 821
37.3.9 ENABLE . . . 822
37.3.10 *FREE . . . 822
37.3.11 *HELP LANMANAGER . . . 822
37.3.12 *LANMAN . . . 822
37.3.13 *LIB . . . 823
37.3.14 *MOUNT . . . 823
37.3.15 *RENAME . . . 823
37.3.16 *SDISC . . . 823
37.3.17 *WIPE . . . 824

38 Network File System 825

39 Alternative File Systems 827
39.1 Acorn System DFS . . . 827
39.1.1 Differences between Acorn DFS and System DFS . . . 827
39.2 Angus Duggan HDFS . . . 827
39.2.1 Catalog . . . 828
39.2.2 Differences between Acorn DFS and HDFS . . . 828
39.3 Opus DDOS . . . 830
39.4 Watford DFS . . . 831
39.4.1 Catalog format . . . 831
39.4.2 Differences between Acorn DFS and WDFS . . . 831
39.5 Watford DDFS . . . 833
39.6 Watford 62 Disk File System . . . 833
39.7 Telesoft File System . . . 834
39.8 IEEE File System . . . 834

The Commodore C64 835

40 Kernal **837**
40.1 Spelling of Kernal 837

41 Memory Layout **839**
41.1 Colour RAM 839

42 Serial Bus **841**
42.1 Types of Device 841
42.2 Addressing 842
42.3 Serial Bus Cable 842

The Sinclair ZX Spectrum 843

43 Spectrum Keyboard **845**

44 Spectrum Memory Map **847**
44.1 Memory Map Overview 847
44.2 Memory Contention 848
44.3 128K and +2 Memory Map 849
44.3.1 Switching RAM and ROM's 849
44.4 Spectrum +2A and +3 Memory Map 851
44.4.1 Normal Mode 851
44.4.2 Special Mode 853
44.4.3 Plus 3 specifics 853

45 File Formats **855**
45.1 TAP Format 855
45.1.1 Block Wrapper 855

46 Machine Code **857**

Modern Day 859

47 Reference Files 861
47.1 Available XML files . . . 861
47.1.1 Fonts . . . 861
47.1.2 Datasets . . . 862
47.2 Character Sets . . . 863
47.2.1 hex . . . 863
47.2.2 op . . . 864
47.2.3 ascii . . . 864
47.2.4 bbc . . . 864
47.2.5 spectrum . . . 864
47.2.6 zx81 . . . 864
47.3 Processor instructions . . . 865
47.3.1 6502 . . . 865
47.3.2 Z80 . . . 866
47.4 Screen Resolutions . . . 868
47.4.1 mode . . . 868
47.4.2 colours . . . 869
47.4.3 textWidth and textHeight . . . 869
47.4.4 graphicsWidth and graphicsHeight . . . 869

48 Reference API Files 871
48.1 Acorn MOS API's . . . 871
48.1.1 hex attribute . . . 872
48.1.2 addr and op attributes . . . 872
48.1.3 vAddr and vName attributes . . . 872
48.1.4 name attribute . . . 872
48.1.5 synopsis element . . . 873
48.2 Acorn MOS OSBYTE . . . 874
48.2.1 hex attribute . . . 874
48.2.2 name attribute . . . 874
48.2.3 bbcb attribute . . . 874
48.2.4 bbcmaster attribute . . . 874
48.2.5 electron attribute . . . 875
48.2.6 var attribute . . . 875
48.2.7 synopsis element . . . 875

48.3 Acorn MOS OSWORD 875
48.3.1 hex attribute 876
48.3.2 name attribute 876
48.3.3 synopsis element 876
48.4 Spectrum 48K ROM API's 877
48.4.1 hex attribute 877
48.4.2 op attributes 877
48.4.3 name attribute 877
48.4.4 synopsis element 878

49 Reference XML Format 879
49.1 char - Define a character 879
49.2 font - Define a bitmap font 881
49.2.1 Attributes 881
49.2.2 Child elements 881
49.2.3 Example font 882
49.2.4 Using a font 882
49.3 fontFile - Define a bitmap font source 883
49.3.1 Attributes 883
49.4 format - Define a machinecode instruction format 884
49.4.1 op attribute 884
49.4.2 description element 884
49.4.3 type element 884
49.4.4 type.label attribute 884
49.4.5 type.byte element 885
49.4.6 type.byte.label attribute 885
49.5 formatType - Define a bit type used within a format 887
49.5.1 caption attribute 887
49.5.2 label and text attributes 887
49.5.3 altLabel and altText attributes 887
49.5.4 style element 888
49.5.5 byte element 888
49.6 generate - Define the content to generate 890
49.6.1 dir attribute 890
49.6.2 Child elements 890
49.7 hexgrid - Generate Hex Grids 892
49.7.1 ascii attribute 893
49.7.2 escape attribute 893

49.7.3 pattern attribute . . . 893
49.7.4 file element . . . 894
49.7.5 match element . . . 894
49.7.6 xaxis and yaxis elements . . . 895
49.7.7 style element . . . 895
49.7.8 default element . . . 896
49.7.9 legend element . . . 897
49.7.10 $(*variable*) expansion . . . 897
49.8 opcode - Define a dataset entry . . . 899
49.8.1 hex attribute . . . 899
49.8.2 op attribute . . . 899
49.8.3 mode attribute . . . 899
49.8.4 opCode attribute . . . 900
49.8.5 group attribute . . . 900
49.8.6 Custom Attributes . . . 900
49.8.7 synopsis element . . . 900
49.9 opcodeFigure - Generate tables for each entry . . . 901
49.9.1 filename attribute . . . 901
49.9.2 Example . . . 901
49.10 opcodeFormat - Generate instruction formats . . . 903
49.10.1 file attribute . . . 903
49.10.2 margin attribute . . . 903
49.10.3 caption element . . . 903
49.10.4 label element . . . 903
49.10.5 index element . . . 904
49.10.6 prefix element . . . 904
49.11 opcodeList - Generate table of all entries . . . 906
49.11.1 filename attribute . . . 906
49.11.2 Example . . . 906
49.12 opcodeFigure and opcodeList common features . . . 907
49.12.1 group attribute . . . 907
49.12.2 caption element and noCaption attribute . . . 907
49.12.3 allowEmpty . . . 907
49.12.4 layout attribute . . . 907
49.12.5 clearPage attribute . . . 908
49.12.6 chapter, section and subsection elements . . . 908
49.12.7 tableType attribute . . . 910
49.12.8 vfill attribute . . . 910

49.12.9 align attribute . . . 910
49.12.10 Table rules . . . 911
49.12.11 Volume 0 index . . . 911
49.12.12 prefix and suffix Elements . . . 912
49.12.13 indexOn Element . . . 912
49.12.14 orderBy Element . . . 913
49.12.15 field Elements . . . 914
49.12.16 dynamic fields . . . 916
49.13 proc - root element . . . 918
49.13.1 proc attribute . . . 918
49.13.2 Child elements . . . 918

APPENDICES 919

A 6502 Instructions 921
A.1 6502 Instruction Grid . . . 936

B Z80 Instructions 939
B.1 Z80 Instruction Grid . . . 941
B.1.1 Instructions with no prefix . . . 942
B.1.2 Instructions after `0xCB` . . . 944
B.1.3 Instructions after `0xDD` . . . 946
B.1.4 Instructions after `0xED` . . . 948
B.1.5 Instructions after `0xFD` . . . 950

C Z80 Instruction Decoding Tables 953
C.1 How to use the decoding tables . . . 953
C.1.1 Examples . . . 953
C.2 What use are these tables? . . . 955
C.3 Instruction Decoding tables . . . 956
C.3.1 Bit types . . . 957
C.3.2 Decoding tables . . . 959

D Machine Operating System 965
D.1 Calls and Vectors . . . 965
D.1.1 Errata . . . 965
D.2 OSByte calls . . . 966
D.2.1 General OSBYTE calls . . . 966

D.2.2 System Variable OSBYTE calls . . . 966
D.2.3 Table layout . . . 966
D.3 OSWord calls . . . 973
D.4 Zero-page use . . . 974
D.4.1 Current Language . . . 974
D.4.2 Econet Workspace . . . 974
D.4.3 NMI Workspace . . . 974
D.4.4 File System Workspace . . . 974
D.4.5 VDU Workspace . . . 974
D.5 System Variables . . . 977
D.5.1 Updating via OSBYTE . . . 977
D.5.2 Table reference . . . 977
D.6 Machine type returned by INKEY and OSBYTE 0x81 . . . 983
D.7 Keyboard key codes . . . 986
D.7.1 Duplicate keys . . . 986
D.7.2 Break key . . . 987
D.7.3 Shift keys . . . 987
D.8 VDU code summary . . . 991
D.8.1 VDU 23 . . . 992
D.8.2 VDU Erratta . . . 992
D.9 BBC Master 128 CMOS RAM, BBC Master Compact EEPROM format . . . 993

E Filesystem Commands 997
E.1 BBC Model B . . . 997
E.2 Acorn Electron . . . 997
E.3 BBC Master Series . . . 997

F BBC Micro and Acorn Electron Ports 999
F.1 RGB . . . 999
F.2 RS-423 . . . 999
F.3 Cassette . . . 1000
F.4 Analogue Input . . . 1001
F.5 Econet . . . 1002
F.5.1 SPROW Ethernet . . . 1002
F.6 Acorn Electron Cartridge Slot . . . 1003
F.7 Master 128 Cartridge Slot . . . 1004
F.8 Master Compact Cartridge Slot . . . 1005

F.9 Electron, Master and Compact cartridge slot notes 1006
F.9.1 +5V supply 1006
F.9.2 −5V supply 1006
F.9.3 Address bus 1006
F.9.4 Data bus 1006
F.9.5 phi clocks 1007
F.9.6 $\overline{ROMOE}$ 1007
F.9.7 *ROMQA* 1007
F.9.8 $CSR\overline{W}$ 1008
F.9.9 $R\overline{W}$ 1008
F.9.10 $\overline{RDY}$ 1008
F.9.11 *NPGFC* and *NPGFD* 1008
F.9.12 *CLOCK* 1009
F.9.13 $\overline{MADET}$ 1010
F.9.14 *LPSTB* 1010
F.9.15 $PB_{5...7}$ User Port 1010
F.9.16 Audio I/O 1011
F.10 MM2114N 4Kb 1024x4 static RAM 1011

G Commodore Ports 1013
G.1 Serial Bus 1013
G.2 User Port 1014
G.3 RS-232 Port 1015
G.4 Expansion Connector 1016
G.4.1 BA Bus Available 1017
G.4.2 DMA Direct Memory Access 1017
G.4.3 Notes 1017

H Sinclair ZX-81 Ports 1019
H.1 Edge Connector 1019

I Spectrum Keyboard Matrix 1021

J Sinclair Spectrum Ports 1023
J.1 Monitor 1023
J.2 RS-232/Midi 1024
J.3 AY-3-8912 Register 14 1024
J.4 Keypad 1025
J.5 AY-3-8912 Register 14 1025

J.6 Amstrad Lightgun 1025
J.7 Spectrum 48K Edge Connector 1026
J.8 Spectrum 128K Edge Connector 1026

K Text screen planning sheets 1027
K.1 80x32 text screen planning sheet 1028
K.2 40x32 text screen planning sheet 1030
K.3 40x25 text screen planning sheet 1032
K.4 32x24 text screen planning sheet 1034
K.5 20x32 text screen planning sheet 1036

L Graphics Planning Sheets 1039
L.1 4x4 Character design sheet 1040
L.2 Character design sheet 1042
L.3 C64 Monocolour Sprite design sheet 1044
L.4 C64 Multicolour Sprite design sheet 1046

INDICES 1049

List of Figures 1051

List of Tables 1055

List of Listings 1063

Bibliography 1067

Glossary 1069

Basic 1123

BBC Basic 1124

Sinclair Basic 1125

TI Basic 1126

6502 Microprocessor Family 1128

Z80 Microprocessor **1131**

BBC Basic Assembler Index **1134**

BeebASM Assembler Index **1135**

Lancaster Assembler Index **1137**

Errata **1141**

Alphabetical Index **1142**

Page intentionally blank

Second Edition

This is the Second Edition of my notes about Developing for Microcomputers.

The First Edition was the various websites I ran from 1996 until 2025. These had a range of domain names over the years, from `www.demon.co.uk` in 1996 followed by `retep.org.uk` in 1998, `retep.org` from 2000 and finally `area51.dev` which went offline in early 2025.

In January 2025 I started to convert the old web content into LaTeX with the intention of making this available online as a set of volumes, each one consisting of a major topic like Assembly Language, Basic or details about specific systems and platforms.

I felt that it was time to make a printable book version of this content, potentially in an actual dead-tree format, so that a lot of this knowledge would not be potentially lost.

In either case, I intend to publish this to at least the Internet Archive as well as GitHub.

– Peter Mount, June 2, 2025

Completion

Since I started this project in early 2025, I've succumbed to an illness which has been progressively getting worse over time. This has caused me, amongst other things, to become permanently offline.

Whilst this has been happening, I've been attempting to complete this work during the periods where I'm able to.

As I write this, it's now the end of March 2026, and I'm still progressing slowly through this, although progress is becoming slower as things progress. I'm not certain when, or if I will complete this, or how I will make this available without access to the Internet.

– Peter Mount, March 30, 2026

I've got to the point where I'm making a copy of this public, although it's been really difficult the last few weeks. This is far from complete, but it has got to the point where the content that has been done might be useful to others.

I'll probably take a break as my health is not good right now, but resume in a few months if I feel better.

– Peter Mount, May 14, 2026

Basic

Table of contents

1 Introduction to BASIC **3**

2 Basic Programming Concepts **5**
2.1 Program Lines 5
2.2 Listing a program 6
2.3 Running a program 6
2.4 Storing and Retrieving a program 7
2.5 Variables 8
2.6 Decisions 9
2.7 Program Flow 10
2.8 Inline Data 11

3 BBC Basic **13**
3.1 Variables 14
3.2 Commands 15
3.3 Statements 18
3.4 Functions 20

4 Commodore Basic **23**

5 Sinclair Basic **25**
5.1 Statements 25

6 TI Basic **35**
6.1 Variables 35
6.2 Commands 37
6.3 Statements 38
6.4 Functions 50
6.5 Subprograms 51

1 Introduction to BASIC

During the Microcomputer age of the early 1980's, the first language everyone began programming with was BASIC, or more specifically Beginners All-purpose Symbolic Instruction Code. The reason for this was that for the majority of microcomputers of that age, it was the default[1] programming language available on the machine.

BASIC, which actually dates from the late 1960's, had common features which were shared across most machines. However, as things were not standardized, each manufacturer's version of BASIC had some variations with each other. Many had extra features to showcase what their machines could do that others could not, whilst some were limited by the manufacturers wanting to keep costs down.

What we will go through here is a general outline of the BASIC language covering the core features shared by every version of the language.

We will then cover the differences that exist between the original basic with those which were provided by microcomputers of the age. Specifically:

- Commodore Basic for the C64
- BBC Basic[2] for the BBC Microcomputer and Acorn Electron
- MSBasic by Microsoft for the original 8-bit computers of the 1974's
- Sinclair Basic for the Sinclair ZX-81, Spectrum and Spectrum 128K
- TI Basic for the TI99/4A

This is not an exhaustive list, but cover's the versions of BASIC I have used in the past.

1: Some early computers did not have a language, or some like the Jupiter Ace had Forth instead of BASIC

2: BBC Basic was not limited to just the Acorn range of machines

Page intentionally blank

Basic Programming Concepts 2

On most machines, when they were first powered on you were presented with a command prompt. At that prompt you could type in a command in BASIC and the computer would act immediately on that command.

2.1 Program Lines

If, however, you started the line with a number, then that line would instead, be stored in memory. The number, or line-number, would determine where in your program the line would be placed, and the order the command would be executed.

Some versions of basic would allow a program to be written without line numbers.

Those versions came later, whilst here we are concerned with the original versions of BASIC which took line numbers.

It didn't matter which order you typed the lines, as it was the line number which determined the order of the program. If you typed a program with the numbers in a different order, they got stored in the correct order.

For example, if you typed the following:

```
20 PRINT a
10 LET a=10
```

Then the order it would be stored would be the following:

```
10 LET a=10
20 PRINT a
```

If you typed in a line whose line number already existed, then the new one you just typed would replace the existing one.

Similarly, entering just the line number would delete any existing line with that number.

2.1.1 Multiple statements per line

Many of the early versions of BASIC only allowed a single statement per line.

However, by the time of the mainstream microcomputers, you could enter multiple statements per line. These were separated from each other with the **:** colon[1] character.

1: Some versions used ; semi-colon instead

2.2 Listing a program

While you are typing in your program, you could use the **LIST** command to show your program. Some variants of BASIC allow you to type a line number to look at just that line, or a range to list line numbers between a pair of line numbers.

2.3 Running a program

To run your program, you would type in **RUN**. This would then run your program from the beginning.

Some versions of BASIC allowed you to include a line number. When this was done then the program would start from that line number rather than the beginning.

Subsection 2.4.1 on the facing page covers how to load and run a program automatically.

2.4 Storing and Retrieving a program

This can vary on a per-system basis, but in general you can save your program using the **SAVE** command, providing it with a file name. This would then cause the computer to save it to either Cassette, or if you had them, to floppy disk.

Similarly, you could load a previously saved program using the **LOAD** command, which loaded the named[2] program from cassette or floppy disk.

2: Some versions allowed "" or similar to mean the first program found rather than having to type in a filename.

2.4.1 Retrieving and running a program

Normally to load and run a program you would have to type two commands, one to do the **LOAD** then **RUN** to run it. Some versions of BASIC supported the ability to do both in a single command. The way this was done differed between each flavour of BASIC, so it is not standard.

With Sinclair Basic you have to set the line number to start execution at when you save the program as Listing 2.1 shows.

```
1 10 PRINT "HELLO WORLD"
2 20 GOTO 10
3 SAVE "myprog" LINE 10
```

Listing 2.1: Sinclair Basic save program with auto execute

BBC Basic is simpler by providing the **CHAIN** command. This was identical to **LOAD** except that it would automatically run the program from the start. This command could also be used inside one program to automatically run another – a technique that was common in implementing menus on compendium discs.

2.5 Variables

In BASIC, variables are represented by a letter or name. Each version of BASIC has its own syntax for variables.

2.5.1 Numeric Variables

A numeric variable is usually represented solely by a letter. These can be either integer or floating point - although some early versions of basic only supported integers and could not perform floating point arithmetic.

2.5.2 Integer Variables

Integer variables were usually handled as numeric variables, however some versions of Basic allowed for variables which were pure integers. Specifically, they were not floating-point so could not have a decimal place. These were usually a letter followed by a % character. *e.g.* P%

2.5.3 String Variables

It's very rare to a string variable in BASIC without the $ symbol

String variables, *e.g.* those representing string values, usually start or end with a $ symbol.

For example A$, or $A.

2.6 Decisions

2.6.1 IF THEN

Most programs have to make decisions, where the program flow changed according to some condition within the program. In BASIC this was performed with the **IF** statement.

IF *condition* **THEN** *true-statement*

If the *condition* is true then the *true-statement* is executed. In many versions this is just a line number or can contain just a **goto** statement, but in others this can be one or more statements.

Listing 2.2: Simple guess the number

```
10 INPUT "Enter a number", b
20 IF b==42 THEN PRINT "That is correct": STOP
30 IF b<42 THEN PRINT "That is too small, try
    again"
40 IF b>42 THEN PRINT "That is too big, try
    again"
50 GOTO 10
```

2.6.2 IF THEN ELSE

Later versions of BASIC expanded the **IF** statement with an optional **ELSE** clause. When the **ELSE** clause was present then the *false-statement* would be executed when the *condition* was false.

IF *condition* **THEN** *true-statement* **ELSE** *false-statement*

Listing 2.3: Simple guess the number with ELSE

```
10 INPUT "Enter a number", b
20 IF b==42 THEN GOTO 40 ELSE PRINT "That is
    incorrect, try again"
30 GOTO 10
40 PRINT "That is correct"
50 STOP
```

2.7 Program Flow

2.7.1 Goto

A common statement is **GOTO** which causes the program to jump unconditionally to another line.

The classical "Hello World" is an example of this:

Listing 2.4: Classic Hello World in Basic

```
10 PRINT "Hello World"
20 GOTO 10
```

2.7.2 Subroutines

A common feature in most versions of Basic is that of the subroutine. This is a block of code which you can call from anywhere within the program with it's starting line number. This is usually done with the **GOSUB** statement.

The program will then run from that location, until it executes a **RETURN** statement. At that point it will return to the location of the **GOSUB** statement which called it.

Modifying Listing 2.2 on the preceding page to use a subroutine we get:

Listing 2.5: Simple guess the number

```
10 INPUT "Enter a number", b
20 IF b==42 THEN PRINT "That is correct": STOP
30 IF b<42 THEN GOSUB 100
40 IF b>42 THEN GOSUB 100
50 GOTO 10
100 PRINT "Number incorrect, please try again"
110 RETURN
```

2.8 Inline Data

Many versions of BASIC allow for data to be included within the program. This data is usually values required by the program which are better suited to be included separately.

For example, it could be as simple as a list of names, or a sequence of numbers to constants required for some complex polynomial.

This is usually provided by the **DATA** statement which stores the data and the **READ** statement which reads one or more entries into a set of variables.

```
10 FOR n=1 TO 6
20 READ d
30 PRINT d
40 NEXT n
50 DATA 1,1,2,3,5,7
```

Listing 2.6: Data example 1

```
10 READ a,b,c
20 PRINT a+b+c
30 DATA 10,20,30
```

Listing 2.7: Data example 2

Page intentionally blank

3 BBC Basic

BBC Basic is the basic which came with the BBC Microcomputer and later the Acorn Electron. The BBC Master 128 series came with an updated version.

This version of basic was popular due to its use within schools in the UK due to the BBC's Computer Literacy project, hence the BBC being part of its name.

This was so popular that other machines got a version of BBC Basic. *e.g.* the Wren Executive[1] was a Z80 CP/M based machine which had BBC Basic on floppy.

1: I had one of these machines in the early 1990's. Sadly I don't know what my Parent's did with it.

BBC Basic is still available today for modern machines. Even parts of the test suite for the ARM processors are written in BBC Basic - the first ARM processors were actually prototyped in it!

Some of the main differences from Generic[2] Basic are:

2: Generic Chapter 2 on page 5

- Multiple statements per line. Each statement is separated by a colon : character.
- **IF** statement can have multiple statements after the **THEN**
- **IF** statement can also have an optional **ELSE** clause which holds code to invoke when the condition is false
- Named procedures, which are subroutines but named rather than called by their line number. They can also have local variables and parameters.

3.1 Variables

Variable names are case sensitive, hence `A1` and `a1` are different variables.

By convention variables are in uppercase.

In BBC Basic there are three kinds of variables:

3.1.1 Number variables

These hold a numeric value which can be either an integer or a real number. They must start with a letter but can be multiple characters long and subsequent characters can be digits.

e.g. `A`, `A1` and `a1` are all valid numeric variables.

3.1.2 String variables

String variables hold textual content. These follow the same naming rules as numbers, however they must also end with the `$` character.

e.g. `A$`, `A1$` and `a1$` are all valid string variables.

3.1.3 Integer variables

BBC Basic supports true integer variables. These are variables which can hold only an integer and never a real number. These follow the same naming rules as numbers, however they must also end with the `%` character.

e.g. `A%`, `A1%` and `a1%` are all valid integer variables.

Some variables like `P%` and `O%` have special meaning with the built-in assembler.

3.2 Commands

In general, entering lines into the program is as standard. There are a few additions to this specific to BBC Basic.

3.2.1 AUTO
Automatically enter line numbers

This command cannot be placed in a program.

It can only be used at the command line.

Syntax **AUTO** [*start*] [,*step*]

Description Automatically enter line numbers when typing in a program.

To exit AUTO mode, simply press ESCAPE.

start If provided then the line numbers start with the provided number.
If not provided then it starts with 10.

step If provided then the line numbers will step with this amount.
If not provided then it adds 10 to each line number.

Note Providing step with no start is valid, the start will be 10.. blank line

3.2.2 DELETE
Delete lines from the program

This command cannot be placed in a program.

It can only be used at the command line.

Syntax **DELETE** *start* , *end*

Description Delete all lines in the program whos line numbers are between the given start and end numbers.

3.2.3 RENUMBER
Renumber the lines in a program

This command cannot be placed in a program.

It can only be used at the command line.

Syntax **RENUMBER** [*start*] [,*step*]

Description Renumber all lines within the program.

start If provided then the program will start with the specified line number. If not provided then it starts with 10.

step If provided then the line numbers will step with this amount. If not provided then it adds 10 to each line number. note

This command is useful if you have written a long program and the line numbers are all over the place. You can issue this command and the program will have the line numbers looking neat. It also enables for room to be made between line numbers so you can insert additional statements.

3.3 Statements

3.3.1 IF
Execute code if condition true

Syntax **IF** *condition* **THEN** *code...* [***ELSE*** *code...*]

Description Test a condition.

condition true If the condition is true then execute the code after the **THEN** statement.

condition false If the condition is false then, if there is an **ELSE** statement then invoke that.

3.4 Functions

3.4.1 ABS
Absolute Value

Syntax ABS (*value*)

Description Returns the absolute, or positive version of a value.

Example The Absolute value of −9.75 is 9.75 but the absolute value of 4.56 is 4.56

See also SGN[3]

3: SGN ?? on page ??

Another example is taking the difference between two values.

If you have $A = 7$ and $B = 10$ then the difference $A - B$ would be positive if A is larger than B, or negative if A is less than B. With ABS the result is always positive.

3.4.2 ACS
Arc-Cosine

Syntax ACS (*value*)

Description Returns the arc-cosine a value. The returned angle is in radians, but can be converted to Degrees with the DEG function.

See also ASN[4], ATN[5], COS[6], DEG[7], SIN[8], TAN[9]

4: ASN ?? on page ??
5: ATN ?? on page ??
6: COS ?? on page ??
7: DEG ?? on page ??
8: SIN ?? on page ??
9: TAN ?? on page ??

```
10 X=ACS(Y)
1205 angle=DEG(ACS(0.5678))
```

3.4.3 ADVAL
Analogue to digital converter value

Syntax ADVAL (*channel*)

Description Sample one of the four available analogue channels or the fire button status

The BBC Micro has a joystick port on the back of the machine[10]. This port, which is a D15 female port, provides two digital and four analogue channels, the latter can be sampled by the ADVAL function.

10: The Acorn Electron also has this port with the Plus-1 extension unit.

Figure F.6 on page 1001 shows the pin layout for this port.

The channel number defines what the ADVAL function returns.

Channels 1 ... 4

These returns the values of each of the four available analogue channels.

Channel 0

ADVAL (0) returns the state of the two fire buttons also on that port.

It also includes what channel was the last one to be updated. This is because it takes time to sample the port, and it does them in sequence.

Table 3.1: ADVAL(0) AND 3

X	Button Pressed
0	None
1	Left
2	Right
3	Both

X = ADVAL (0) AND 3

returns a value 0 ... 3 which Table 3.1 shows the combinations.

X = ADVAL (0) DIV256

returns the number of the last analogue to digital conversion. If this returns 0 then no channel has yet completed a conversion.

Negative channels

When a negative channel is passed to the **ADVAL** function it returns details about various buffers. *e.g.***ADVAL** (-1) returns the number of characters in the keyboard buffer.

See OSBYTE `0x80` below for more details.

OSBYTE `0x80` (128)

11: OSBYTE `0x80` **??** on page ??

The **ADVAL** function is in reality the OSBYTE `0x80`[11] call provided by the operating system. This call takes the channel number like the **ADVAL** function in the X register and returns the results in the X and Y registers.

X=1 ... 4 This is the same as **ADVAL** (X) returning the 16 bit value in X and Y

X=0 This is the same as **ADVAL** (0). On return, Y holds the ADC channel last converted whilst bit's 0 and 1 hold the fire button status.

X < 0 When X is negative, it's the same as **ADVAL** with a negative channel. **??** on page ?? shows the buffer whos status is returned.

Commodore Basic 4

Page intentionally blank

5 Sinclair Basic

Sinclair Basic, as used on the Sinclair Spectrum machines, follows general Basic.

Some of the main differences from Generic[1] Basic are:

- Multiple statements per line. Each statement is separated by a colon : character.
- **IF** statement can have multiple statements after the **THEN**

1: Generic Chapter 2 on page 5

5.1 Statements

In this section we describe each individual statement that exists within Sinclair Basic.

5.1.1 DATA
Store data to be read later

Syntax **DATA** *value* [, *value* ...]

Description Stores data in a program which will be read later by the **READ** statement. see also

5.1.2 FOR perform loop

Syntax **FOR** *variable* = *start* **TO** *end* [***STEP*** *step*]
...
NEXT *variable*

Description Execute a loop setting the supplied variable with a value. At the start of the loop the variable will be given the start value, and the loop will continue until it reaches the end value.

STEP If provided, the step value is used to increment the variable between iterations. If the step value is negative then the variable is decremented. If the step value is absent then 1 is used.

NEXT The **NEXT** statement marks the end of the loop. When it is encountered, BASIC will end the current itteration and start a new one from where the **FOR** statement is located. If the loop terminates, then the program will continue with the statement after the **NEXT**. item

```
10 FOR m=0 TO 6
20 FOR n=0 TO m
30 PRINT m;":";n;" ";
40 NEXT n
50 PRINT
60 NEXT m
```

Listing 5.1: Sinclair Basic For loop with nesting

5.1.3 IF
Execute code if condition true

Syntax **IF** *condition* **THEN** *code...*

Description Test a condition.

condition true If the condition is true then execute the code after the **THEN** statement.

condition false If not met then continue execution from the following line.

5.1.4 GO SUB
Call Subroutine

Syntax **GO SUB** *lineno*

Description Call a subroutine at the specified line number.

When the subroutine returns via the **RETURN** statement then the statement after the **GO SUB** will be executed.

See also **RETURN**[2]

2: **RETURN**
Subsection 5.1.9 on page 34

Unlike most versions of BASIC, Sinclair Basic has the statement as two words instead of the usual single "go-sub". This doesn't really matter much as on the pre-128K machines the statement is entered as a single key-press.

5.1.5 GO TO
Jump to line

Syntax **GO TO** *lineno*

Description Jump to the specified line number.

This is used to perform simple flow control within the program. Once the jump has been performed, unlike **GO SUB**, you cannot return to the original location.

Unlike most versions of BASIC, Sinclair Basic has the statement as two words instead of the usual single "goto". This doesn't really matter much as on the pre-128K machines the statement is entered as a single key-press.

5.1.6 LET
Set Variable

Syntax **LET** *variable* = *expression*

Description Execute an *expression* and set the result to the named variable.

5.1.7 READ
Read data into Variable

Syntax **READ** *variable* [, *variable* ...]

Description Read a value from a **DATA** statement in the program into one or more variables.
see also

5.1.8 RESTORE
Set start of data

Syntax **RESTORE** [*lineno*]

Description This sets the line where the next data the **READ** statement will read from. If the line number is not provided then this sets the read location to the beginning of the program. see also

5.1.9 RETURN
Return from Subroutine

Syntax **RETURN**

Description Returns from a subroutine which was called by **GO SUB**.

See also **GO SUB**[3]

3: **GO SUB** Subsection 5.1.4 on page 29

6 TI Basic

TI-Basic was the version of BASIC which was included with the TI-99/4A microcomputer from 1981.

Unlike most computers of that era, the TI was actually a 16-bit machine whose processor was based on Texas Instruments mini computers. Sadly for the TI, Texas Instruments crippled the machine with some of its architectural decisions, so it ran software slowly if it ran without the memory expansion.

As for TI-Basic, there were two versions in use.

TI-Basic was the version on the machine, and TI-Extended Basic which was a cartridge and provided additional functionality not available to the onboard version of basic.

One of the main differences between TI-Basic and TI-Extended[1] Basic is that in TI-Basic you can only have a single statement per line. In TI-Extended Basic you can have multiple statements per line.

[1] *Texas Instruments TI-99/4A Users Reference Guide*, (Texas Instruments 1982) P.59

6.1 Variables

In TI-Basic there are two kinds of variables:

6.1.1 Numberic variables

These hold a numeric value which can be either an integer or a real number. They must start with a letter but can be multiple characters long and subsequent characters can be digits.

e.g. `A` and `VALUE` are all valid numeric variables.

6.1.2 String variables

String variables hold textual content. These follow the same naming rules as numbers[1], however they must also end with the $ character.

e.g. `A$` and `NAME$` are all valid string variables.

1: numbers Subsection 6.1.1 on the preceding page

6.2 Commands

6.3 Statements

6.3.1 DATA
Store data inside the program

Syntax **DATA** value [, *value...*]

Description Stores data in the program

See also **READ**[2], **RESTORE**[3]

2: **READ** Subsection 6.3.12 on page 48

3: **RESTORE** Subsection 6.3.14 on page 48

The **DATA** statement stores data inside the program which can be read by the **READ** statement.

DATA statements can be located anywhere within the program but the order is important as they are read sequentially.

6.3.2 DISPLAY
DISPLAY to the screen

Syntax **DISPLAY** value [, *value...*]

Description Display to the screen

See also **DISPLAY**

The **DISPLAY** statement is identical to **PRINT** when writing to the screen. However, it cannot be used to write on any other device than the screen.

Other than that restriction, **DISPLAY** follows the same rules as **PRINT**.

TI-Extended Basic

Extended basic also provides the **DISPLAY** ...**USING** statement..

6.3.3 END
End the program

Syntax **END**

Description Terminates the program.

See also **STOP**[4]

In TI-Basic this can be used interchangeably with the **STOP** statement.

4: **STOP** Subsection 6.3.15 on page 49

6.3.4 FOR perform loop

```
>NEW
>100 FOR I=1 TO 5
>140 PRINT I;
>150 NEXT I
>160 END
>RUN
 1  2  3  4  5
 ** DONE **
```

```
>NEW
>100 FOR I=5 TO 1
    STEP -1
>140 PRINT I;
>150 NEXT I
>160 END
>RUN
 5  4  3  2  1
 ** DONE **
```

Syntax **FOR** variable = start **TO** end [***STEP*** *step*]

Description Execute a loop setting the supplied variable with a value. At the start of the loop the variable will be given the start value, and the loop will continue until it reaches the end value.

STEP If provided, the step value is used to increment the variable between iterations. If the step value is negative then the variable is decremented. If the step value is absent then 1 is used.

NEXT The **NEXT** statement marks the end of the loop. When it is encountered, BASIC will end the current itteration and start a new one from where the **FOR** statement is located. If the loop terminates, then the program will continue with the statement after the **NEXT**. item

See also **NEXT**[5]

5: **NEXT** Subsection 6.3.9 on page 43

Changing the control variable

If the variable is changed during the loop then the number of iterations performed by the loop will be affected.

Start after End

If the start value is after the end value (or before if the step increment is negative) then the loop will not be performed.

Evaluation order

In TI-Basic, the expressions for *start*, *end* and *step* are evaluated before the variable is first set on the first run. So if one of these is set based on the same variable used by the loop, then that value is set once at the start but is not affected by the variable changing afterwards.

?? on page ?? shows a loop which will be evaluated 5 times as the end value is set before the variable I is redefined.

```
>NEW
>100 I=5
>110 FOR I=1 TO I
>140 PRINT I;
>150 NEXT I
>160 END
>RUN
 1  2  3  4  5
** DONE **
```

6.3.5 GOTO
Jump to line

Syntax **GOTO** lineno

GO TO lineno

Description Jump to the specified line number.

This is used to perform simple flow control within the program.

See also **ON-GOTO**[6]

Although TI-Basic uses the standard **GOTO** statement, it does also support a space between the two so **GO TO** is also valid.

```
>NEW
>100 CALL CLEAR
>110 REM GOTO
    EXAMPLE
>140 PRINT "TEXAS"
>150 PRINT "
    INSTRUMENTS"
>160 GOTO 140
>170 END
```

6: **ON-GOTO**
Subsection 6.3.8 on page 43

6.3.6 IF
Execute code if condition true

Syntax **IF** condition **THEN** *code...* [***ELSE*** *code...*]

Description Test a condition.

condition true If the condition is true then execute the code after the **THEN** statement.

condition false If the condition is false then, if there is an **ELSE** statement then invoke that.

In TI-Basic, if the expression generating condition returns 0 then it's treated as false. Any other numeric value is treated as true.

6.3.7 LET
Set Variable

Syntax [***LET***] variable = *expression*

Description Execute an *expression* and set the result to the named variable.

Note Unlike some versions of BASIC using **LET**, in TI-Basic the **LET** statement is optional.

```
>NEW
>100 LET M=1000
>110 LET C=186000
>120 E=M*C^2
>130 PRINT E
>140 END
>RUN
 3.4596E+13
```

6.3.8 ON-GOTO
Jump to line depending on value

Syntax **ON** expression **GOTO** lineno [, *lineno* ...]

ON expression **GO TO** lineno [, *lineno* ...]

Description Jump to the specified line number based on the result of an expression.

The expression is evaluated and converted into an integer. The integer is then used to select the line number from the list. Integer 1 is the first line number, 2 the second *etc.*

Errors If the value is less than 1 or larger than the number of line numbers present then a `BAD VALUE IN xx` error is issued.

If the line bumber is outside the range of lines in the program then the error `BAD LINE NUMBER` is issued.

See also **GOTO**[7]

Like **GOTO**, this statement allows for a space between the GO and TO.

```
>NEW
>100 REM HOW DOES
    ON-GOTO WORK
>110 INPUT X
>120 ON X GOTO
    130,150,170,190,210

>130 PRINT "X=1"
>140 GOTO 110
>150 PRINT "X=2"
>160 GOTO 110
>170 PRINT "X=3"
>180 GOTO 110
>190 PRINT "X=4"
>200 GOTO 110
>210 END
>RUN
 ? 2
 X=2
 ? 1.2
 X=1
 ? 3.7
 X=4
 ? 6
 * BAD VALUE IN 120
```

7: **GOTO** Subsection 6.3.5 on page 41

6.3.9 NEXT

Syntax **NEXT** variable

Description Marks the end of a **FOR** ...**NEXT** loop.

See also **FOR**[8]

8: **FOR** Subsection 6.3.4 on page 40

6.3.10 INPUT

Syntax **INPUT** [*prompt* **:**] variable [, *variable* ...]

Description Requests input from the user

```
>NEW
>100 INPUT B
>110 PRINT B
>120 END
>RUN
? 25
25

** DONE **
```

The **INPUT** statement is used when entering data via the keyboard whilst the program is running. The statement causes the program to pause until valid data is entered from the keyboard. Although the computer usually accepts up to one program line (4 lines on the screen) for each **INPUT** statement, a long list off values may be rejected by the computer. If you receive the message "`LINE TOO LONG`" after entering an input line, you will need to divide the lengthy **INPUT** statement into at least two separate statements.

Entering the Input Statement

```
>NEW
>100 INPUT "COST OF
     CAR?": B
>110 A$="TAX?"
>120 INPUT A$:C
>130 INPUT "SALES
    "&A$:X
>140 PRINT B;C;X
>150 END
>RUN
 COST OF CAR? 5500
 TAX? 500
 SALES TAX? 500
  5500  500  500

 ** DONE **
```

The optional prompt is a string expression that indicates on the screen the values you should enter at that time. Including a prompt in the **INPUT** statement is optional. When no prompt is provided then the question mark ? is shown followed by a space.

If you use a prompt then the string expression must be followed by a colon. When the computer performs this type of **INPUT** statement it will display the prompt message and then wait for the data to be entered.

Responding to an Input Statement

When an **INPUT** statement is performed, the values corresponding to the variables must be entered in the same order as they are listed in the **INPUT** statement. When you enter the values, they must all be entered in one input line, up to 4 screen lines, with the values separated by commas.

When inputting string values you may enclose the string in quotes. However, if the string you wish to input contains a comma, a leading quote mark, leading spaces or trailing spaces, it *must* be enclosed in quotes.

TI Extended Basic

If extended basic, the input may be made with the **LINPUT** statement without editing.

The **ACCEPT** statement allows input from nearly every position on the screen.

6.3.11 PRINT
PRINT to the screen

Syntax **PRINT** print-list

Description Print data to the screen

See also **DISPLAY**[9]

9: **DISPLAY**
Subsection 6.3.2 on page 38

The **PRINT** statement lets you print numbers and strings on the screen. The print-list consists of:

- print-items, numeric or string expressions which print on the screen
- tab-functions which control print positioning
- print-separators which deal with separating print-items.

Printing Strings

String expressions in the print-list are evaluated to produce a string result. There are no blank spaces inserted before or after a string. If you wish to print a blank space before or after a string you can include it in the string, or insert it separately with quotes.

Printing Numbers

Numeric expressions in the print-list are evaluated to produce a numeric result to be printed. Positive numbers are printed with a leading space (instead of a plus sign) and negative numbers are pritned with a leading minus sign. All numbers are printed with a trailing space.

Print Separators

```
>NEW
>100 A=-26
>110 B=-33
>120 C$="HELLO"
>130 D$="HOW ARE
    YOU?"
>140 PRINT A;B;C$;
    D$
>150 END
>RUN
 -26 -33 HELLOHOW
    ARE YOU?
 ** DONE **
```

Each screen line displayed with the **PRINT** statement has 28 character positions numbered from left to right (1 ... 28). Each line is divided into two 14-character print zones. By using the print-separators and the tab-function, you can control the position of the characters displayed on the screen.

There are three types of print-separators: semicolons, colons and commas. At least one print-separator must be placed between adjacent print-items in the print-list. Multiple print-separators may be used side by side and are evaluated from left to right.

Semicolon

```
>NEW
>100 A=-26
>110 B$="HELLO"
>130 C$="HOW ARE
    YOU?"
>140 PRINT A:B$:C$
>150 END
>RUN
 -26
 HELLO
 HOW ARE YOU?
 ** DONE **
```

A semicolon causes adjacent characters to print side by side with no extra spaces between the values. All numbers are generally printed with a trailing space regardless of the type if print-separator used.

Colon

A colon causes the next print-item to print at the beginning of the next line.

Comma

A comma causes the next print-item to be printed at the beginning of the next print zone. Each line is split into two zones. When the comma is encountered, it starts the second zone. If it was already in the second zone then it starts a new line and uses the first zone on that line.

```
>NEW
>100 A$="ZONE 1"
>110 B$="ZONE 2"
>120 PRINT A$,B$
>130 PRINT A$:,B$,
    A$
>140 END
>RUN
 ZONE 1        ZONE
     2
 ZONE 1
                ZONE
     2
 ZONE 1
 ** DONE **
```

Tab Function

The tab-function works similarly to that of a normal typewriter. With this command you may tabulat the display.

By providing the column number you define the position on the screen line on which you want to place the next character.

The computer evaluates the numeric expression and rounds it to the nearest integer. If the integer is less than one then it is replaced with 1. If it is greater than 28 then it is repeatedly reduced by 28 until it is within the range $1 \leq n \leq 28$.

```
>NEW
>100 A=326
>110 B=79
>120 PRINT A;TAB
    (15);B
>130 PRINT A,B
>140 PRINT A;TAB
    (15),B
>150 PRINT TAB(5);A
    ;TAB(5);B
>170 PRINT A;TAB
    (43);B
>180 END
>RUN
  326           79
  326           79
  326
  79
      326
       79
  326           79
 ** DONE **
```

6.3.12 READ
Read data

Syntax **READ** variable [, *variable* ...]

Description Reads data stored in the program

See also **DATA**[10], **RESTORE**[11]

The **READ** statement allows you to read data stored inside the program in **DATA** statements.

The variable list specifies the variables that are to have values assigned. Each variable name is separated by a comma. The list can contain a mix of numeric and/or string variables.

10: **DATA** Subsection 6.3.1 on page 38

11: **RESTORE** Subsection 6.3.14 on page 48

```
>NEW
>100 FOR I=1 TO 3
>110 READ X,Y
>120 PRINT X;Y
>130 NEXT I
>140 DATA
    22,15,36,52,
    48,96.5
>150 END
>RUN
  22  15
  36  52
  48  96.5
** DONE **
```

6.3.13 REM
Remark

Syntax **REM** *remark*

Description Store a remark within the program.

The **REM** statement allows for a comment to be stored within the program.

6.3.14 RESTORE
Set next DATA to be read

Syntax **RESTORE** [*lineno*]

Description Sets the line to start reading data from

See also **DATA**[12], **READ**[13]

Once a **READ** statement has consumed all data, or there is not enough data available then it fails with the "`DATA ERROR`" error message.

The **RESTORE** statement allows you to reset where it reads data so that it can resume reading.

12: **DATA** Subsection 6.3.1 on page 38

13: **READ** Subsection 6.3.12 on page 48

If the optional line number is provided then the next **READ** statement will start reading from the next **DATA** statement from that line number.

If no line number is provided then the next **DATA** statement read will be the first one in the program.

6.3.15 STOP
Stop the program

Syntax **STOP**

Description Terminates the program.

See also **END**[14]

In TI-Basic this can be used interchangeably with the **END** statement.

14: **END** Subsection 6.3.3 on page 39

6.4 Functions

6.5 Subprograms

TI Basic had the concept of Subprograms which provided access to routines either not easily implemented in TI Basic or in Machine Code.

All of these Subprograms use the **CALL** statement. This statement is followed by the Subprograms name and any parameters in parentheses.

6.5.1 CALL CHAR

Syntax **CALL CHAR** (char-code, "char-definition ")

Description Define the character definition

char-code character code

char-definition Hexadecimal definition of the character

The **CALL CHAR** Subprogram redefines the appearence of a character using a 16 character hexadecimal string. This string, like on other systems, define the 8 bytes defining the appearence, one byte per row of the character.

The char-code can be any value between 128 ... 159 in addition to the standard ASCII characters 32 ... 127.

```
>NEW
>100 CALL CLEAR
>110 CALL CHAR(33,"
    FFFFFFFFFFFFFFFF
    ")
>120 CALL COLOR
    (1,9,6)
>130 CALL VCHAR
    (12,16,33)
>140 GOTO 140
>150 END
>RUN
-- screen clears
```

6.5.2 CALL CLEAR

Syntax **CALL CLEAR**

Description Clears the screen

The **CALL CLEAR** Subprogram is used to clear the entire screen. When it is called, the space character is placed in all positions on the screen.

6.5.3 CALL COLOR

Syntax **CALL COLOR** (charset, foreground, background)

Description Sets the colour of characters when shown on screen

charset The group of characters the colour pertains to

foreground Foreground colour

background Background colour

The **CALL COLOR** Subprogram sets the colour of characters when they are displayed on screen. Unlike how most systems handle character colours, the TI99/4A groups characters into groups.

The default colours are 2 for a *Black* foreground and 1 *transparent* for the background.

When colour 1 *transparent* is used then the screen colour is shown.

Table 6.1 on page 53 shows the possible values for foreground or background.

You also have to define which of the 16 available character sets the colour definition applies to. These sets group characters between codes 32 ... 159 into sets of 8 characters.

So, when the screen has been cleared, as it's filled with spaces (char 32), then the colour definition for set 1 applies to the screen. If it's set to the default then as colour 1 is transparent then the current screen colour is shown.

If the background is anything other than transparent then the screen colour only appears at the top and bottom of the screen.

Code	Colour
1	Transparent
2	Black
3	Medium Green
4	Light Green
5	Dark Blue
6	Light Blue
7	Dark Red
8	Cyan
9	Medium Red
10	Light Red
11	Dark Yellow
12	Light Yellow
13	Dark Green
14	Magenta
15	Grey
16	White

Table 6.1: Colours in TI Basic

6.5.4 CALL GCHAR

Table 6.2: Colour character sets in TI Basic

Code	Char Code
1	32 - 39
2	40 - 47
3	48 - 55
4	56 - 63
5	64 - 71
6	72 - 79
7	80 - 87
8	88 - 95
9	96 - 103
10	104 - 111
11	112 - 119
12	120 - 127
13	128 - 135
14	136 - 143
15	144 - 151
16	152 - 159

```
>NEW
>100 CALL CLEAR
>110 CALL HCHAR
    (1,1,36,768)
>120 CALL GCHAR
    (5,10,X)
>130 CALL CLEAR
>140 PRINT X
>150 END
>RUN
-- screen clears
-- fills with $
    signs
-- screen clears
    again

 36
```

Syntax **CALL GCHAR** (row-number, col-number, variable)

Description Read character on the screen and place it into a variable

row-number Row number on the screen

col-number Column number on the screen

variable Numeric variable to accept the characters ASCII code

See also HCHAR[15], VCHAR[16]

The **CALL VCHAR** Subprogram takes the character at the specified location on the screen and places the ASCII code of that character into a numeric variable.

15: HCHAR
Subsection 6.5.5 on page 55

16: VCHAR
Subsection 6.5.8 on page 59

6.5.5 CALL HCHAR

Syntax **CALL HCHAR** (row-number, col-number, char-code [, *count*])

Description Place and optionally repeat character on screen

row-number Row number on the screen

col-number Column number on the screen

char-code ASCII code of character

count Number of times to repeat

See also GCHAR[17], VCHAR[18]

The **CALL HCHAR** Subprogram places a character anywhere on the screen. If the count parameter is provided then it will repeat that character horizontally that number of times.

17: GCHAR
Subsection 6.5.4 on page 53

18: VCHAR
Subsection 6.5.8 on page 59

6.5.6 CALL SCREEN

Syntax **CALL SCREEN** (colour)

Description Sets the colour of the screen background

colour Background colour

The **CALL SCREEN** Subprogram sets the colour of screen background. On the TI99/4A, the screen colour is shown around the main text content. Unlike many other systems, there is also a transparent colour, so for character sets which have that as it's background colour then the screen colour shows through. The default screen colour is 8 Cyan.

Table 6.1 on page 53 shows the possible values for the screen color.

Transparency

19: CALL COLOUR
Subsection 6.5.3 on page 52

When used by **CALL COLOUR**[19] if the transparent colour is used then the screen colour shows through. This is the default for the background colour.

However, if used for the screen colour, selecting transparent also shows Cyan.



Genlock capability?

I nolonger have the reference to this, but I do remember reading back when the TI was new, that for the screen, if transparent was defined then the video in feed would be used – as the video processor had a genlock capability.

6.5.7 CALL SOUND

Syntax **CALL SOUND** (duration, $freq_1$, vol_1 [, $freq_2$, vol_2 [, $freq_3$, vol_3 [, $freq_4$, vol_4]]])

Description Produce tones of different frequencies

duration Duration of the tine in milliseconds

$freq_n$ Frequency in Hz

vol_n Volume

The TI99/4A has an audio chip which has three tone and one noise generators. It can play all four generators at the same time. The **CALL SOUND** Subprogram provides access to this.

Sound duration

The duration is the period the tone should be played. It is defined in milliseconds and can be a value from $1ms$ to $4250ms$[20].

20: The actual duration can vary as much as $\pm\frac{1}{60}s$

Normally, if the call is made whilst a sound is already playing then the computer will wait for the previous sound to complete before performing the call. If, however, the duration is negative, then the previous sound is stopped and the new one starts immediately.

Volume

The volume can be in the range $0 \ldots 30$ inclusive where 0 is for the loudest, and 30 the quietest.

Frequency

The frequency defines the type of sound produced. For the single noise channel the frequency is a negative value to define the type of noise. Table 6.3 on the facing page shows the possible values.

Table 6.3: TI Noise Characteristics

Frequency	Characteristic	
-1	Periodic Noise	Type 1
-2		Type 2
-3		Type 3
-4		Varies
-5	White Noise	Type 1
-6		Type 2
-7		Type 3
-8		Varies

-4 and -8 varies with the frequency of the third tone specified

When a positive number is used then it represents the frequency of the tone. The frequency[21] can range from $110Hz$ to $44733Hz$, well above human hearing limits.

Table 6.4 on the next page is an incomplete list of musical tones and their associated frequencies.

21: The actual frequency produced can vary by ±10% depending on the frequency requested

Table 6.4: TI Musical Frequencies

Freq	Note	Comment	Freq	Note	Comment
110	A		440	A	*above middle C*
117	A♯, B♭		446	A♯, B♭	
123	B		494	B	
131	C	*low C*	523	C	*high C*
139	C♯, D♭		554	C♯, D♭	
147	D		587	D	
156	D♯, E♭		622	D♯, E♭	
165	E		659	E	
175	F		698	F	
185	F♯, G♭		740	F♯, G♭	
196	G		784	G	
208	G♯, A♭	*erratta*[a]	831	G♯, A♭	
220	A	*below middle C*	880	A	
233	A♯, B♭		932	A♯, B♭	
247	B		988	B	
262	C	*middle C*	1047	C	*above high C*
277	C♯, D♭		1109	C♯, D♭	
294	D		1175	D	
311	D♯, E♭		1245	D♯, E♭	
330	E		1319	E	
349	F		1397	F	
370	F♯, G♭		1480	F♯, G♭	
392	G		1568	G	
415	G♯, A♭		1661	G♯, A♭	
440	A	*above middle C*	1760	A	

Frequencies are in Hz

[a] The original manual had this as B♭ which is incorrect as that's at $233Hz$

6.5.8 CALL VCHAR

Syntax **CALL VCHAR** (row-number, col-number, char-code [, *count*])

Description Place and optionally repeat character on screen

row-number Row number on the screen

col-number Column number on the screen

char-code ASCII code of character

count Number of times to repeat

See also GCHAR[22], HCHAR[23]

The **CALL VCHAR** Subprogram places a character anywhere on the screen. If the count parameter is provided then it will repeat that character vertically that number of times.

22: GCHAR
Subsection 6.5.4 on page 53

23: HCHAR
Subsection 6.5.5 on page 55

Page intentionally blank

Assembly Language

Table of contents

7 Basic Assembly Language Concepts **65**
7.1 Decimal Numbers . . . 65
7.2 Binary Numbers . . . 66
7.3 Hexadecimal Numbers . . . 66
7.4 Octal Numbers . . . 67
7.5 Bytes . . . 68
7.6 Multiple Byte Values . . . 69
7.7 Binary Coded Decimal (BCD) . . . 70
7.8 Signed Numbers . . . 71

8 Flags **75**
8.1 Carry . . . 75
8.2 Zero . . . 76
8.3 Negative . . . 76

9 Program Flow **79**
9.1 Program counter . . . 79
9.2 Jumps . . . 79
9.3 Subroutines . . . 84

10 Boolean Logic **91**
10.1 Logical And . . . 91
10.2 Logical Or . . . 92
10.3 Logical Exclusive OR . . . 93
10.4 Logical Complement . . . 94

11 Shifts and Rotates **97**
11.1 Shift and Rotate Left . . . 98
11.2 Shift and Rotate Right . . . 99
11.3 Multiple Byte Shifts . . . 101

12 Arithmetic **103**
12.1 Increment and Decrement . . . 103
12.2 Addition . . . 104
12.3 Subtraction . . . 109

12.4 Comparisons 114
12.5 Multiplication 115
12.6 Division 117

Page intentionally blank

7 Basic Assembly Language Concepts

Before anyone can begin with Assembly Language they have to know some basic concepts, some of which applies to virtually every programming language in existence.

In this chapter we will cover those basics, from number systems, binary logic and basic arithmetic as used by most processors. Once you have mastered these basic concepts then you can, in theory handle assembly language for most processors, be it the 6502, Z80 families or even modern day processors.

7.1 Decimal Numbers

For most individuals numbers consist of the digits 0 ... 9 in what is known as the Decimal system.

As a decimal number consists of ten digits we also call this as base-10.

For a decimal number, each digit represents a component of the number, each one being a power of ten above the digit to it's right. Hence, for the number 1234, each place stands for 4 plus 3 to the power of ten, 2 to the power of one hundred and 1 to the power of a thousand.

Digit	Power	Value
1	$1 * 10^3$	1000
2	$2 * 10^2$	200
3	$3 * 10^1$	30
4	$4 * 10^0$	4
		1234

For many this is the natural method to represent a number however, in mathematics there is no true sole method to represent a number or do every method require the use of the same symbols to represent said numbers.

7.2 Binary Numbers

In the Binary system there are only two digits, 0 and 1 and is also known as base-2 due to it only having two digits.

Like decimal numbers, each digit is a power above the digit to it's right, however unlike Decimal, each digit is a power of 2.

Figure Figure 7.2 on page 68 Shows the decimal number 102 in its binary form 1100110 and how each bit resolves to its actual value.

7.3 Hexadecimal Numbers

Although Decimal is commonly used for most numeric values and Binary is used when specifying individual bits, it's common for hexadecimal or base-16 numbers to be used.

A hexadecimal digit is a value between 0 and 15 inclusive comprising the normal numeric digits 0 … 9 but then the letters A…F[1] to represent the values 10 … 15.

1: In really old systems from the 1950's the letters at the other end of the alphabet were used instead of A…F

Hexadecimal numbers are more convenient because each digit maps to 4 bits in binary and two digits into a single byte.

Figure 7.1 on the next page shows how Hexadecimal numbers map to both Decimal and Binary.

7.4 Octal Numbers

Another number system to be aware of is Octal. Octal is base-8, where the number is represented by the digits 0 ... 7. In computing this maps to three bits per octal digit.

Although not in general use by the time of the Microcomputer age, Octal was in use with the previous Minicomputer era, especially where Minicomputer instruction sets like the PDP-11 are defined in Octal.

That said Octal is still used in some places in the modern era, although many wouldn't notice this.

File permissions on Linux/Unix or MacOS are defined in Octal - *e.g.* the file permission 0644 for -rw-r−r− is actually in octal.

To differenciate Octal from Decimal, it's common to prefix the number with 0, hence 0644 for the file permissions

Binary	Decimal	Hexadecimal	Octal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	10
1001	9	9	11
1010	10	A	12
1011	11	B	13
1100	12	C	14
1101	13	D	15
1110	14	E	16
1111	15	F	17

Figure 7.1: Hexadecimal conversion chart

7.5 Bytes

Microcomputers and previous generations of digital computers use the Binary number system where each binary digit is represented by a single bit of memory. To make memory useful, these Bits are grouped together to form bytes, where one Byte consists of eight Bits.

A typical microprocessor will then address each byte in memory with each byte representing either an instruction or data.

In this form, a byte can hold an unsigned integer ranging from $0 \ldots 255$ inclusive, or a signed-integer[2] between $-128 \ldots 127$ inclusive.

2: Section 7.8 on page 71

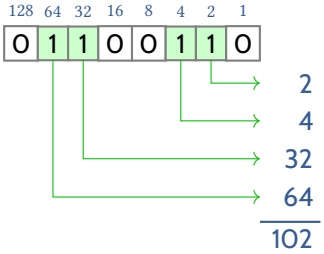


Figure 7.2: Binary representation of a byte

7.5.1 Nibbles

A group of four bits is called a nibble[3] with a byte being formed of two nibbles, a low nibble for bits $0 \ldots 3$ and a high nibble for bits $4 \ldots 7$.

When the term nibble is being used, it is usually written in either hexadecimal or binary-coded-decimal (BCD) as each digit fits into a single nibble.

3: I have also seen Nybble as the spelling, although most literature uses nibble.

7.6 Multiple Byte Values

Bytes can be grouped together to form larger values. For 8-bit microprocessors like the 6502 or Z80 this is commonly a pair of bytes, sometimes called a Word or double-byte, but can be even larger. Addresses are defined as a 16-bit value or a pair of bytes.

For 16-bit microprocessors, they usually work with a word as the normal data size and handling single byte values being the odd one out.

Len	Name	Max Value		Modern
1	Byte	2^8-1	255	uint8
2	Word	$2^{16}-1$	65535	uint16
4	Long	$2^{32}-1$	4294967295	uint32
8	–	$2^{64}-1 \approx 1.844 * 10^{19}$		uint64

Figure 7.3: Capacity of multibyte values

Figure 7.3 on the next page shows the common sizes[4] for multibyte values and the maximum value that can be stored as an unsigned integer.

4: 8-byte values are rare in Retro Computing but common in Modern Computing

7.6.1 Endianess

In computing, Endianesss defines the byte order multiple byte numbers are represented in either memory or in a structure or file. There are two types, Little-Endian and Big-Endian.

For Little-Endian, the least significant byte is first, with each additional byte following it until the most significant byte is included. Big-Endian is the reverse, with the most significant byte first.

For example, if we have a double-word containing the hexadecimal number `0x12345678` then it comprises four bytes, `0x12`, `0x34`, `0x56` and `0x78`. If stored in memory in that order then it is in big endian format.

For Little-Endian, the byte `0x78` is the first in memory, followed by `0x56`, `0x34` and finally `0x12`.

Figure Figure 7.4 on the following page shows this number as it would be stored in memory for both Big-Endian and Little-Endian formats.

The value `0x12345678` stored in memory

	Byte offset			
	+0	+1	+2	+3
Little Endian	`0x78`	`0x56`	`0x34`	`0x12`
Big Endian	`0x12`	`0x34`	`0x56`	`0x78`

Figure 7.4: Byte order

For the majority of Microprocessors, all multibyte operations are Little-Endian in nature, as you will see with arithmetic.

You usually see Big-Endian in Minicomputers and in network frames like TCP/IP.

7.7 Binary Coded Decimal (BCD)

Binary Coded Decimal (BCD) is a special number format provided by some Microprocessors for simple arithmetic where each individual decimal digit is represented not in binary but using a single Nibble per digit.

That is, instead of a byte holding a value from 0 … 255 inclusive, it only holds 0 … 99.

This mode has the benefit that it's easy to convert a BCD encoded number to ASCII or back, however the downsides outnumber the benefits as not all arithmetic operations support BCD and an encoded number takes up more memory than it's binary equivalent.

7.8 Signed Numbers

The majority of operations work on unsigned positive integers. For example a byte can represent any integer between 0 ... 255 whilst a word consists of positive integers 0 ... 65535.

However, there are times when negative integers are required.

There are many methods available to handle this although the most common amongst Microprocessors is the technique called two's complement.

7.8.1 Two's complement

With the Two's complement format, the most significant bit of the number is used to signify the sign. If it's set then the number is negative, if clear then positive.

The rest of the byte(s) represent the actual value.

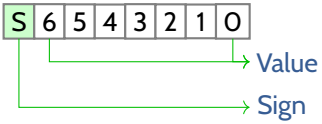


Figure 7.5: Two's complement representation of a byte

All arithmetic operations[5] use two's complement as if you subtract below 0 you get negative numbers.

e.g. $0 - 1 =$ `0xFF` which is either 255 as an unsigned integer but -1 in two's complement.

In most modern languages[6] as well as many from the retro days, uint is unsigned values but int is a signed value which uses two's complement.

5: The only time I've come arithmetic not using two's complement is BCD arithmetic

6: C is thought to be a moden language but actually dates from the 1970's

Figure Figure 7.6 on page 72 shows common values for an 8-bit signed number, whilst figure Figure 7.7 shows the minimum and maximum values supported by signed two's complement numbers of different byte sizes.

Decimal	Hexadecimal	Binary
−128	0x80	1000 0000
−127	0x81	1000 0001
−126	0x82	1000 0010
⋮	⋮	⋮
−3	0xFD	1111 1101
−2	0xFE	1111 1110
−1	0xFF	1111 1111
0	0x00	0000 0000
1	0x01	0000 0001
2	0x02	0000 0010
⋮	⋮	⋮
126	0x7E	0111 1110
127	0x7F	0111 1111

Figure 7.6: 8-bit two's complement numbers

Len	Name	Minimum Value		Maximum Value		Modern
1	Byte	-2^7	−128	2^7-1	127	int8
2	Word	-2^{15}	−32768	$2^{15}-1$	−32767	int16
4	Long	-2^{31}	−2147483648	$2^{31}-1$	2147483647	int32
8	–	-2^{63}	$\approx -9.223 * 10^{18}$	$2^{63}-1$	$\approx 9.223 * 10^{18}$	int64

Figure 7.7: Capacity of multibyte two's complement numbers

7.8.2 One's complement

Although not used as often as Two's complement, One's complement is similar for positive integers where they are represented by their usual binary value.

However, for negative values, the value is the binary value for the positive value complemented so that every 0 and 1 are switched.

This means that a signed byte holds ± 127 rather than $-127 \ldots 128$ in two's complement.

$$+3 = \texttt{00000011}_2$$
$$-3 = \texttt{11111100}_2 \qquad \text{One's complement}$$
$$+3 = \texttt{11111101}_2 \qquad \text{Two's complement}$$

Negation

Negation is simply invert all the bits by applying an exclusive or with `0xFF`:

$$\neg\, 10_{10} = \texttt{0x0A}_{16} \;\; \texttt{0xFF}_{16} = \texttt{0xF5}_{16}$$

Two values of zero

There is an additional complication with one's complement where we have two possible values of zero, where:

$$+0 = \texttt{00000000}_2 = \texttt{0x00}_{16} \qquad \text{but}$$
$$-0 = \texttt{11111111}_2 = \texttt{0xFF}_{16} \qquad \text{hence}$$
$$+0 \neq -0$$

It is for this reason why most arithmetic instructions operate with Two's complement instead.

Modern floating point arithmetic does support ± 0 but that is out of scope of this book

7.8.3 Signed bit only

A third format is where the most significant bit is set for negative values but the value portion is identical to the positive value.

$$10 = \texttt{00001001}_2 = \texttt{0x0A}_{16}$$
$$-10 = \texttt{10001001}_2 = \texttt{0x8A}_{16}$$

Like One's complement this does mean we have the senario of $+0 \neq -0$.

$+0 = \texttt{00000000}_2 = \texttt{0x00}_{16}$ but
$-0 = \texttt{10000000}_2 = \texttt{0x80}_{16}$ hence
$+0 \neq -0$

This is rarely seen for integer values but is common for floating point where they are usually represented by a sign, mantissa and exponent - the latter two always being the same regardless of the sign.

8 Flags

All Microprocessors have a flag register which holds the status of the microprocessor and its operations. The name can differ between processors as can the name and number of each flag but the general purpose of these flags remain the same.

Table 8.1 on page 77 lists the flags available by microprocessor.

8.1 Carry

One of the most common flags is the Carry Flag. This is used by instructions to carry a bit between operations.

For Addition[1], the Carry Flag will be set when the result of an addition overflowed its register size so when the next addition-with-carry operation occurs the value of the Carry flag is included in that operations result.

For Subtraction[2], the Carry Flag is used as a borrow so that, if it's not set then an additional one is removed from a subtraction-with-carry operations result.

Bit Shift operations also use the Carry flag, moving any bit shifted off the register into the flag whilst some operations will use the carry flag as the value to shift into the register.

1: Addition Section 12.2 on page 104

2: Subtraction Section 12.3 on page 109

8.2 Zero

The other most common flag is the Zero Flag. This flag is usually set whenever some operation's result is zero.

For example, a Subtraction from a register causes the final result to be 0 then the Zero flag is set. If not then it's clear.

Comparison's also set this flag when two values are equal. This is because a comparison operation is actually a subtraction operation but the result other than the flags is discarded.

Other operations can also set this, some load operations from memory can affect the flags, as can a Addition operation[3].

3: For Addition to result with a zero result requires either two's-complement or negative values to be used, or the unsigned value to wrap around

8.3 Negative

Most processors support the Negative flag. This flag is usually set whenever the result would be negative if it's two's-complement[4] value is negative.

4: Subsection 7.8.1 on page 71

Specifically, for 8-bit values if bit-7 of the result is set then the negative flag is also set. With 16-bit values then bit-15 is used.

Table 8.1: Status Flags by Processor

	Microprocessor			
Flag	6502	65816[1]	Z80	Description
Carry	c	c	c	Carry flag
Zero	z	z	z	Result is zero
Negative	n	n	s[5]	Result is Negative
Break	b			BRK instruction
Decimal	d	d		Decimal (BCD) mode
Emulation		e[2]		Emulation mode
Index Register		x		Index Register Select
Half Carry			h[2]	Half Carry
$\overline{IRQ}$	i	i		Disable Interrupts
Memory		m		Memory/Accumulator select
Parity			p/v[4]	Parity
Overflow	v	v	p/v[4]	Overflow
Subtract			n[2]	Subtract

1. 65816 flags are for when running in Native (e=0) mode.
   When in Emulation (e=1) mode then the 6502 flags are used.
2. The 65816 e flag is not accessible from user code other than via the XCE instruction.
3. The H and N flags are used for BCD arithmetic and cannot be tested.
4. The P/V flag's use depends on the instruction being executed but is a single bit in the register.
5. On the Z80 the negative flag is known as the sign flag.

Page intentionally blank

9 Program Flow

Program Flow represents the sequence of instructions the processor executes as part of a program. This flow consists of one of a group of possible patterns: Sequential, branches, loops, subroutines and interrupts. How these are implemented are common on all processors with relatively minor differences in the instruction syntax.

9.1 Program counter

Every Microprocessor has a PC Register. This register holds the current address that the current instruction is located.

Normally once that instruction has been executed then the counter is incremented to point to the next instruction. However, some instructions will cause the counter to be set to an alternate address in memory.

For example a Jump instruction simply sets the next address, whilst a Jump-to-subroutine instruction will save the next address to the stack before pointing to the subroutine. When the subroutine completes, the address is removed from the stack and execution resumes from the next instruction.

9.2 Jumps

Jump instructions implement unconditional jumps in a program where the next instruction is in another location in memory away from the current location. They can be used to implement loops where code is repeatedly executed, sometimes indefinitely.

9.2.1 Relative Jumps

Relative jumps are unconditional jumps which use an offset from the current location to calculate the destination address.

As these use relative rather than an absolute address, the code can be placed at any location in memory without need to be reassembled with the new address. The downside however, is that on most 8-bit microprocessors, these offsets are only 8-bit, so the destination can only be within ±127 bytes of the current location.

As such, they are usually used for short loops.

Z80

The offset can be within the range -126…+129 bytes of this instruction's address.

On the Z80, an unconditional relative jump is implemented by the JR instruction.

6502

The original 6502 microprocessor does not support unconditional relative jumps.

65C02

The offset can be within the range -128…+127 bytes of this instruction's address.

On the 65C02 and later, the BRA provides this instruction.

9.2.2 Long Relative Jumps

Most Relative Jumps are limited to destination addresses within a short distance of the instruction's address. However, some processors support longer relative jumps.

65816

The 65816 processor has the BRL instruction which accepts a 16-bit relative offset which can be used to jump to any address within the current program-bank.

The offset can be within the range $-32768 \cdots +32767$ bytes of this instruction's address.

9.2.3 Absolute Jumps

An Absolute Jump is a jump to any address accessible within the processor's address map. For 8-bit processors like the 6502 and Z80 this is a 16-bit address enabling a total of 64K of addressable memory.

The 6502 JMP and Z80 JP instructions implement an absolute jump.

9.2.4 Banked Jumps

Some processors extend the address space by implementing memory banks.

This is not the same as banking techniques used on some machines like the Spectrum 128K, C64 or BBC Microcomputer to expand available memory. Those machines use external hardware to switch blocks of memory within the normal 64K address space to different memory devices.

65816

The 65816 in native mode is an example of banked jumps.

Here it has a program bank register which enables the processor to address 16 Megabytes of memory using 256 64K blocks.

So it's JMP instruction[1] has a long versions of some of its address modes to change the destination bank as well as address within that bank.

1: Some assemblers call this JML to differenciate it from the original JMP instrction

9.2.5 Conditional Branches

Conditional Branches are jumps which happen based on the state of the processor's flags.

In most instances, conditional branches are relative jumps so have the same restrictions of how far the destination of the branch is. However, some processors like the Z80 also allow branches to absolute addresses.

Table 9.1 on page 82 shows the instructions available per condition on the 6502 and Z80 processors.

Table 9.1: Conditional Branches by Processor

		Z80	
Condition	6502	Relative	Absolute
Equality	BEQ *rel*	JR Z, *rel*	JP Z, *abs*
Inequality	BNE *rel*	JR NZ, *rel*	JP NZ, *abs*
Carry Set	BCS *rel*	JR C, *rel*	JP C, *abs*
Carry Clear	BCC *rel*	JR NC, *rel*	JP NC, *abs*
Negative	BMI *rel*	JR M, *rel*	JP M, *abs*
Positive	BPL *rel*	JR P, *rel*	JP P, *abs*
Overflow Clear	BVC *rel*		
Overflow Set	BVS *rel*		
Parity Even		JR PE, *rel*	JP PE, *abs*
Parity Odd		JR PO, *rel*	JP PO, *abs*

Z80

As well as the conditional instructions in table Table 9.1, the Z80 also supports conditions against other instructions.

CALL a subroutine can also be conditional. This is described in Section 9.3 on the next page.

DJNZ is an instruction which decrements the B register and then, if the result is not zero, perform a relative jump.

The DJNZ instruction is effectively a combination of the DEC B and JR NZ instructions but is one byte shorter and executes quicker than the separate instructions.

It is mainly intended for use in loops.

RET from a subroutine can also be conditional. This is described in Section 9.3 on the following page.

9.3 Subroutines

A subroutine is a block of code that can be called repeatedly from various parts of a program which can return control back to the original caller.

Parameters can be passed to these subroutines via either registers or by setting predefined memory locations before making the subroutine call.

9.3.1 Calling a Subroutine

When a subroutine is called, the processor stores the return address onto the stack, then sets the Program Counter to the subroutines address so that the processor then starts executing the subroutine.

6502

On the 6502 family the JSR (Jump to SubRoutine) instruction is used to call a subroutine.

The target address can be anywhere within the 64K address space.

65816

The 65816 is identical to the 6502 except that it also provides two additional instructions.

JSR *long* A long version of the JSR[2] instruction which also takes the target memory bank containing the subroutine. This routine must, however, return using the new RTL instruction and not RTS.

2: Some assemblers accept JSL as an alternate name for this instruction

JSR (*addr*,X) This instruction will add the current X index register to the given address and then use that address to obtain the subroutine's address. This instruction is usually used for lookup tables.

Z80

The Z80 provides two instructions for calling subroutines.

CALL *addr* The Main one is CALL which takes an address anywhere within the 64K address space.

RST *num* The other is the RST instruction which is a single byte instruction and can call a subroutine located at one of a fixed set of addresses at the lower end of memory. This instruction is usually used for implementing Operating System calls or interrupt handlers.

9.3.2 Recursion

Subroutines are free to call themselves. It is up to the individual routine to manage it's state so that calling itself does not corrupt the routine itself.

The number of levels of recursion available is down to the size of the available stack and the number of bytes per call used in making the subroutine call. The size of the stack is processor dependent.

9.3.3 Returning from a Subroutine

To exit a subroutine and return control to the calling program requires the return-from-subroutine instruction. This instruction pulls the return address from the stack and places it into the Program Counter so that the processor then resumes execution from the calling program.

6502

RTS instruction is used for any subroutine called by JSR.

65816

RTS instruction is used for any subroutine called by normal JSR instruction like the 6502.

RTL instruction is used for any subroutine called by the long version of JSR. This is because the long version of JSR uses three bytes on the stack whilst the normal JSR uses only two.

Z80

RET instruction is used to return from a subroutine.

9.3.4 Conditional Subroutine Calls

Some processors allow for conditional subroutine calls where the call is made only if a condition is met. For those processors which do not support this, then this can be emulated with a conditional branch.

Z80

On the Z80 both the CALL and RET instructions accept the same conditions as Conditional Branches.

When this form of the instructions are used then the instruction is only executed if the required condition is met.

If the condition is not met, then execution continues with the next instruction in memory.

Table Table 9.2 on page 87 shows the instructions available per condition.

Condition	Call	Return
Equality	CALL Z, *rel*	RET Z
Inequality	CALL NZ, *rel*	RET NZ
Carry Set	CALL C, *rel*	RET C
Carry Clear	CALL NC, *rel*	RET NC
Negative	CALL M, *rel*	RET M
Positive	CALL P, *rel*	RET P
Parity Even	CALL PE, *rel*	RET PE
Parity Odd	CALL PO, *rel*	RET PO

Table 9.2: Conditional Subroutine Calls for the Z80

6502 Conditional JSR

The 6502 Family does not support conditional subroutine calls, but this can be emulated with the conditional branch instructions.

The trick here, however, is to use the branch not to call the subroutine but to skip the JSR call to it. This requires the use of the anti-branch instruction to the action you require.

For example, if you want to call a subroutine if a condition is equal, instead of using BEQ, you use BNE instead to skip the actual JSR instruction.

Listing 9.1: Conditional JSR on 6502

```
1  ; Z80 code
2      CP      #0x0D
3      CALL    Z, NEWLINE  ; Call NEWLINE if A==0
       x0D
4      ; Resume routine
5
6  ; 6502 Equivalent
7      CMP     #0x0D
8      BNE     SKIP        ; Skip if A<>0x0D
9      JSR     NEWLINE     ; Call NEWLINE when A=0
       x0D
10 .SKIP
11     ; Resume routine
```

Here:

Line 7 Compares accumulator with 0x0D
Line 8 BNE skips the actual JSR so that
Line 9 JSR only executes when the accumulator equals 0x0D.
Line 10 Execution then resumes

6502 Conditional RTS

Similarly the 6502 doesn't support conditional RTS but it's easily emulated by using a branch to an RTS instruction located elsewhere:

Listing 9.2: Conditional RTS on 6502

```
1  ; Z80 code
2  .ROUTINE1       ; Simple example to loop 10
       times
3      LD  B, #10
4  .LOOP
5      DEC B       ; Decrement counter at loop
       start
6      RET Z       ; Return when B is 0
7      ; Do something useful here
8      JR  LOOP    ; Restart the loop
9
10 ; 6502 Equivalent
11 .ROUTINE1       ; Simple example to loop 10
       times
12     LDX # 10
13 .LOOP
14     DEX         ; Decrement counter at loop
       start
15     BEQ ENDLOOP ; Return if X is zero
16     ; Do something useful here
17     JMP LOOP    ; Restart the loop
18 .ENDLOOP        ; Return
19     RTS
```

Here:

Line 14 Decrements the X register
Line 15 BEQ emulates RET Z by branching to the RTS on line 19
Line 16 Remainder of the loop goes here
Line 17 Jump back to line 13 for the next iteration

Page intentionally blank

10 Boolean Logic

One of the fundamental operations for any Microcomputer is to compare two values be it at a numeric level or at the bit level.

10.1 Logical And

A logical AND operation will set a bit in the output only if that bit is set in both values of the operation.

This follows the truth table in Figure Figure 10.1 on page 91.

Figure 10.2 shows an example of the instruction:

```
; Pseudo code
LOAD    A, #0xDA
AND     #0x45       ; A = 0x42
```

	128	64	32	16	8	4	2	1	
	1	1	0	1	1	0	1	0	0xDA
AND	0	1	0	0	0	1	1	0	0x45
Result	0	1	0	0	0	0	1	0	0x42

Figure 10.2: Logical AND operation

For most processors, the operation affects the following flags:

Z Set when the result is zero, otherwise clear.
N Set if the most significant[1] bit of the result is set.

Figure 10.1: AND truth table

Memory \ Accumulator	0	1
0	0	0
1	0	1

1: Bit-7 for 8-bit values and Bit-15 for 16-bit values

10.2 Logical Or

A logical OR operation will set a bit in the output if that bit is set in either value of the operation.

This follows the truth table in Figure Figure 10.3 on the following page.

Figure 10.4 shows an example of the instruction:

```
1 ; Pseudo code
2 LOAD    A, #0xDA
3 OR      #0x45       ; A = 0xDE
```

Figure 10.3: OR truth table

Memory \ Accumulator	0	1
0	0	1
1	1	1

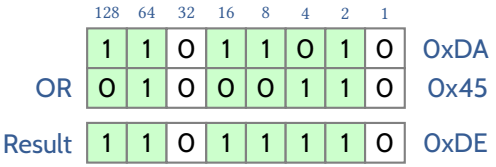


Figure 10.4: Logical AND operation

For most processors, the operation affects the following flags:

Z Set when the result is zero, otherwise clear.
N Set if the most significant[2] bit of the result is set.

2: Bit-7 for 8-bit values and Bit-15 for 16-bit values

10.3 Logical Exclusive OR

A logical Exclusive OR operation will set a bit in the output if that bit is set in one value but clear in the other.

This follows the truth table in Figure Figure 10.5 on the facing page.

Figure 10.6 shows an example of the instruction:

```
1 ; Pseudo code
2 LOAD    A, #0xDA
3 XOR     #0x45       ; A = 0x9C
```

	128	64	32	16	8	4	2	1	
	1	1	0	1	1	0	1	0	0xDA
XOR	0	1	0	0	0	1	1	0	0x45
Result	1	0	0	1	1	1	0	0	0x9C

For most processors, the operation affects the following flags:

Z Set when the result is zero, otherwise clear.
N Set if the most significant[3] bit of the result is set.

Figure 10.5: Exclusive OR truth table

Memory \ Accumulator	0	1
0	0	1
1	1	0

Figure 10.6: Logical Exclusive Or operation

3: Bit-7 for 8-bit values and Bit-15 for 16-bit values

10.3.1 Exclusive OR on the 6502

On the 6502 family, the Logical Exclusive Or operation is implemented by the EOR instruction.

10.3.2 Exclusive OR on the Z80

On the Z80, the Logical Exclusive Or operation is implemented by the XOR instruction.

10.4 Logical Complement

A logical Complement operation will invert the bit's in a value, so if a bit was set then it is cleared and vis-versa.

Figure fig:asm:logic:complement:logic shows an example of the instruction:

```
; Pseudo code
LOAD    A, #0xDA
NOT     A           ; A = 0x25
```

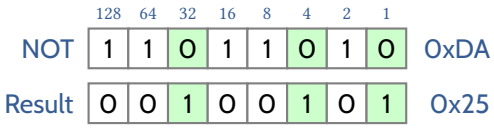


Figure 10.7: Logical Complement operation

For most processors, the operation affects the following flags:

Z Set when the result is zero, otherwise clear.
N Set if the most significant[4] bit of the result is set.

4: Bit-7 for 8-bit values and Bit-15 for 16-bit values

10.4.1 Complement on the 6502

The 6502 processor does not have a NOT instruction, however it can be emulated with the EOR instruction

```
; 6502 code
EOR #0xFF       ; Complement the Accumulator
```

10.4.2 Complement on the 65816

Like the 6502, the 65816 does not have a NOT instruction. When the processor is in native mode and is using 16-bit accumulators then the value required is `0xFFFF`. Otherwise, it follows the 6502 and requires `0xFF` in both native and emulated modes.

```
1 ; 65816 code
2 EOR #0xFFFF     ; When using 16-bit accumulator
3 EOR #0xFF       ; When using 8-bit accumulator
```

10.4.3 Complement on the Z80

The Z80 has two instructions to deal with complementing the accumulator.

These are described in full in Section 19.2 on page 255.

CPL Complement Accumulator

The first is **CPL**[5] which complements the Accumulator by inverting the bits. This is the same as **XOR 0xFF**[6].

5: **CPL** Section 22.12 on page 293
6: **XOR 0xFF** Section 22.67 on page 390

NEG Two's complement Accumulator

The other instruction is **NEG**[7] which negates the Accumulator in a two's complement safe manner. This is effectively $A = 0 - A$, so $neg(0) = 0$ but $neg(1) = -1$(`0xFF`).

7: **NEG** Section 22.35 on page 335

Page intentionally blank

11 Shifts and Rotates

Although we show 8-bit shifts in these examples, 16-bit processors can perform shifts 16-bits at a time instead of 8.

Shift and Rotate operations moves each bit of a value into the adjacent bit on either the left or right of their current position.

A shift to the left means that each bit is shifted into the next-higher-order bit, effectively multiplying the value by two.

A shift to the right moves each bit into the next lower-order bit. This divides the value by two with the bit shifted into the carry flag the remainder.



Whats the difference between a Shift and Rotate operation?

The difference between a shift and a rotate is that in a shift the bit is removed from the value never to return, whilst in a rotate it reappears back at the other end, sometimes via the Carry Flag.

11.1 Shift and Rotate Left

11.1.1 Rotate Left with Carry

The carry flag is placed into bit-0, whilst the rest of the value is shifted left one bit. The original most significant bit, usually bit-7, is placed into the carry flag.

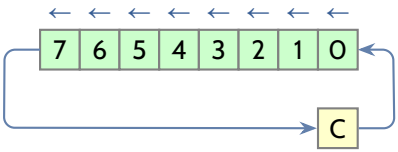


Figure 11.1: Rotate Left with Carry

11.1.2 Rotate Left with Branch Carry

This is similar[1] to a normal rotate left, except that the most significant bit is placed into both the carry and bit-0.

1: I've only seen this with the Z80's **RLC** instruction.

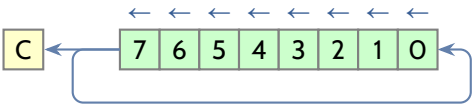


Figure 11.2: Rotate Left with Branch Carry

11.1.3 Shift Left

A shift-left is similar to a rotate-left operation, except that a 0 is placed into bit-0, whilst the most significant bit is placed into the carry flag.

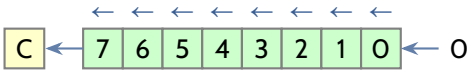


Figure 11.3: Arithmetic Shift Left

In modern languages like **C** or **Go™** , the << operation would be this one.

11.2 Shift and Rotate Right

11.2.1 Rotate Right with Carry

The carry flag is placed into the most significant bit, usually bit-7, whilst the rest of the value is shifted right one place. The original bit-0 is placed into the carry flag.

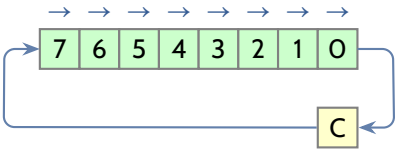


Figure 11.4: Rotate Right with Carry

11.2.2 Rotate Right with Branch Carry

Like it's left equivalent, this operation is rare[2]. This is similar to a normal rotate right, except that bit-0 is placed into both the carry and the most significant bit, usually bit-7.

2: I've only seen this with the Z80's **RRC** instruction.

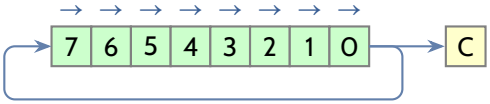


Figure 11.5: Rotate Right with Branch Carry

11.2.3 Shift Right

There are two versions for this operation and this is not supported by all processors. In both versions, it shifts the value right one bit with bit-0 being placed into the carry flag.

Two's complement shift right

In this version, whilst all the bits are shifted right one place, the most significant bit, usually bit-7, is left unchanged. This has the effect that if the original value was negative in two's complement terms, then the result remains negative.

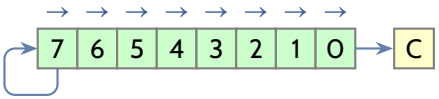


Figure 11.6: Arithmetic Shift Right

The Z80 supports this with the **RRC**[3] and **RRCA**[4] instructions.

The 6502 does not have this operation.

3: **RRC** Section 22.57 on page 368
4: **RRCA** Section 22.56 on page 367

One's complement shift right

In this version a 0 is placed into the most significant bit of the value. I've not seen this version on any microprocessor but is being mentioned here for completeness.

The downside is that if the value was originally negative, it is not after the operation.

It can be emulated as a normal rotate right with carry with the carry flag being cleared before the rotate.

In modern languages like **C** or **Go™** , the >> operation would be this one.

11.3 Multiple Byte Shifts

11.3.1 Shift multiple left

Shift left by performing the shift on each byte in sequence. The Carry flag is used to pass bit-7 to bit-0 on the next byte.

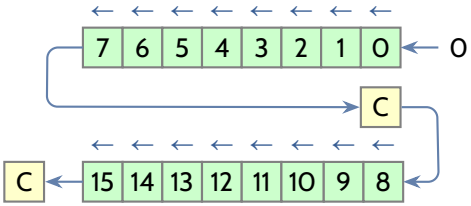


Figure 11.7: Shift left multiple

```
; 6502                  Z80
ASL 0x70                LD  HL, 0x70
ROL 0x71                SLA (HL)
INC HL
RL  (HL)
```

Rotate left uses the same instruction for each byte. Bit-0 will receive the initial carry flag.

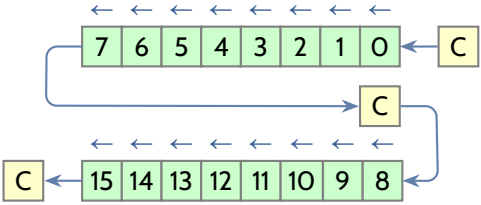


Figure 11.8: Rotate left multiple

```
; 6502                  Z80
ROL 0x70                LD  HL, 0x70
ROL 0x71                RL  (HL)
INC HL
RL  (HL)
```

11.3.2 Shift multiple right

Similar to shifting left except we process the bytes in the reverse order. The Carry flag is used to pass bit-0 to bit-7 on the next byte.

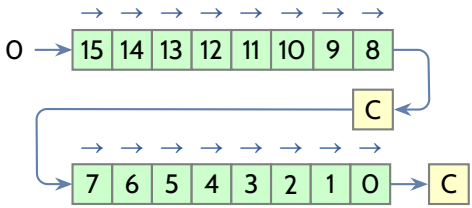


Figure 11.9: Shift right multiple

```
; 6502                  Z80
LSR 0x71                LD  HL, 0x71
ROR 0x70                SRA (HL)
DEC HL
RR  (HL)
```

Like rotating-left, here we use the same instruction for each byte.

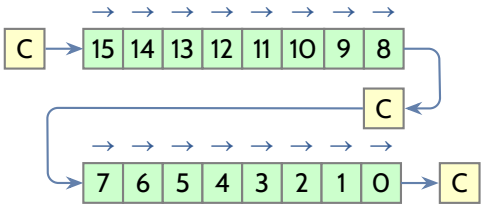


Figure 11.10: Rotate right multiple

```
; 6502                  Z80
ROR 0x71                LD  HL, 0x71
ROR 0x70                RR  (HL)
DEC HL
RR  (HL)
```

12 Arithmetic

As well as boolean logic and shifts, microprocessors can handle simple arithmetic.

Usually this is limited to integer addition and subtraction, with increment, decrement and comparisons being special cases of those.

Multiplication and Division is usually left for implementation in software with few hardware implementations being available before the 16-bit processors where some had math coprocessors.

12.1 Increment and Decrement

The simplest arithmetic instructions are Increment and Decrement. These instructions simply add or subtract one from a register or memory location.

12.1.1 6502

The 6502 family supports the increment and decrement instructions against either the processor's registers and to a memory location.

Table 12.1 on page 103 lists the available instructions on the 6502.

Instruction	Description
DEC	Decrement memory or accumulator
DEX	Decrement X index register
DEY	Decrement Y index register
INC	Increment memory or accumulator
INX	Increment X index register
INY	Increment Y index register

Table 12.1: 6502 Increment and Decrement instructions

12.1.2 65816

The 65816 is identical to the 6502, except that in native mode then it can be 16-bit dependent on the m and x flags.

12.1.3 Z80

The Z80 provides the INC and DEC instructions which can increment or decrement either a single 8-bit register, a 16-bit register pair or a memory location. The memory location can only be referenced via the HL, IX or IY registers and not directly.

12.2 Addition

Addition comes in two forms, addition with carry and addition without carry.

Not all Microprocessors support both, but it is usually easy to implement the other on those systems that do not support the missing operation.

In general use, both operations will set the Carry Flag if the result of the addition overflowed. However, the Addition with Carry operation will also add the Carry Flag to the result if it was initially set, whilst the without-carry operation would ignore the Carry Flag.

For example, if we were to calculate $3+7$, then we would use something like the following pseudocode:

```
1 ; Pseudo code
2 LOAD    A, #3
3 ADD     #7
4 STORE   A, 0x70
```

Here we loaded some register with the number 3, added 7 using add without carry and then stored it somewhere.

If we needed to include the current carry flag in the result then we would use the ADC instruction instead of the ADD instruction.

12.2.1 Multibyte addition

Now if this was a multiple-byte operation we would do the same for the second byte, but in this case use the add with carry instruction so that the carry from a previous instruction would cause the result to be correct when the earlier bytes overflow.

For example:

```
; Pseudo code
LOAD    A, 0x70 ; Load least significant byte
ADD     #7      ; add 7, ignoring Carry flag
STORE   A, 0x70 ; Store result
LOAD    A, 0x71 ; Load next byte
ADC     #0      ; Add 0 + Carry
STORE   A, 0x71 ; Store result in second byte
```

As you can see, in this instance we added 7 to a 2-byte word, so we add 7 without carry to the first byte then 0 to the second byte as we treat 7 as a 16-bit number, and it's upper 8-bits are 0. We still add it if it's 0 as we need to add the carry from the previous add.

Also, the first add MUST be an addition without carry instruction but any subsequent ones must be addition with carry, hence using ADD first then ADC second. If we had double-word (4 byte) values here then we would ADD the first byte, then ADC for each of the subsequent bytes in sequence:

```
; Pseudo code
LOAD    A, 0x70 ; Load least significant byte
ADD     #7      ; add 7, ignoring Carry flag
STORE   A, 0x70 ; Store result
LOAD    A, 0x71 ; Load next byte
ADC     #0      ; Add 0 + Carry
STORE   A, 0x71 ; Store result in second byte
LOAD    A, 0x72 ; Load next byte
```

```
ADC     #0      ; Add 0 + Carry
STORE   A, 0x72 ; Store result in second byte
LOAD    A, 0x73 ; Load next byte
ADC     #0      ; Add 0 + Carry
STORE   A, 0x73 ; Store result in second byte
```

12.2.2 Addition on the 6502

The 6502 family only supports addition with carry with the **ADC**[1] instruction. It does not have an addition without carry instruction, but you can emulate it by preceding the ADC instruction with **CLC**[2] (Clear Carry).

1: **ADC** Section 16.1 on page 134

2: **CLC** Section 16.16 on page 153

For example, adding 3 to 7 for a single byte for the 6502:

```
; 6502 code
CLC         ; Clear Carry
LDA #3      ; Load 3
ADC #7      ; Add 7
STA 0x70    ; Store result
```

Similarly, for multiple byte addition:

```
; 6502 code
CLC         ; Clear Carry
LDA 0x70    ; Load first byte
ADC #7      ; Add 7
STA 0x70    ; Store first byte
LDA 0x71    ; Load second byte
ADC #0      ; Add 0 + Carry from first byte
STA 0x71    ; Store second byte
```

Note we use CLC before the first addition to emulate ADD but not for the subsequent ones.

12.2.3 Addition on the 65816

The 16-bit version of the 6502 supports 16-bit addition with carry when it's running in native mode, so the following would add 7 to the 16-bit value at 0x70 and 0x71:

```
; 6502 code
CLC         ; Clear Carry
LDA 0x70    ; Load 16-bit value from 0x70
ADC #7      ; Add 7
STA 0x70    ; Store 16-bit result
```

12.2.4 Addition on the Z80

The Z80 supports both **ADD**[3] (addition without Carry) and **ADC**[4] (addition with carry) instructions,

```
; Z80 code
LD  A, (0x70)   ; Load 7-bit value from 0x70
ADD A, #7       ; Add 7 without carry
LD  (0x70), A   ; Store result
```

The Z80 also supports 16-bit addition when using the 16-bit register pairs:

```
; Z80 code
LD  HL, (0x70)  ; Load 16-bit value from 0x70
LD  BC, #7      ; Set BC to 7
ADD HL, BC      ; Add BC without carry
LD  (0x70), HL  ; Store 16-bit result
```

3: **ADD** Section 22.2 on page 278

4: **ADC** Section 22.1 on page 276

12.3 Subtraction

Like Addition, subtraction comes in two forms, subtraction with carry and subtraction without carry. Not all Microprocessors support both, but it is usually easy to implement the other on those systems that do not support the missing operation.

In general use, both operations will set the Carry Flag if the result of the addition overflowed. However, the Subtraction with Carry operation will also subtract the carry flag to the result if it was initially set, whilst the without-carry operation would ignore the Carry Flag.

For example, if we were to calculate $3 - 7$, then we would use something like the following pseudocode:

```
; Pseudo code
LOAD    A, #7
SUB     #3
STORE   A, 0x70
```

If we needed to include the current Carry Flag in the result then we would use the SBC instruction instead of the SUB instruction.

12.3.1 Multibyte subtraction

Now if this was a multiple-byte operation we would do the same for the second byte, but in this case use the add-with-carry instruction so that the carry from a previous instruction would cause the result to be correct when the earlier bytes overflow.

For example:

```
; Pseudo code
LOAD    A, 0x70 ; Load least significant byte
SUB     #7      ; Subtract 7, ignoring Carry
    flag
STORE   A, 0x70 ; Store result
LOAD    A, 0x71 ; Load next byte
SBC     #0      ; Subtract 0 + Carry
STORE   A, 0x71 ; Store result in second byte
```

As you can see, in this instance we subtracted 7 from a 2-byte word, so we subtract 7 without carry to the first byte then 0 to the second byte as we treat 7 as a 16-bit number, and it's upper 8-bits are 0. We still subtract it if it's 0 as we need to subtract the carry from the previous subtraction.

Also, the first add MUST be a subtraction without carry instruction but any subsequent ones must be subtraction with carry, hence using SUB first then SBC second. If we had double-word (4 byte) values here then we would SUB the first byte, then SBC for each of the subsequent bytes in sequence:

```
; Pseudo code
LOAD    A, 0x70 ; Load least significant byte
SUB     #7      ; Subtract 7, ignoring Carry
    flag
STORE   A, 0x70 ; Store result
LOAD    A, 0x71 ; Load next byte
SBC     #0      ; Subtract 0 + Carry
STORE   A, 0x71 ; Store result in second byte
LOAD    A, 0x72 ; Load next byte
SBC     #0      ; Subtract 0 + Carry
STORE   A, 0x72 ; Store result in second byte
```

```
11 LOAD    A, 0x73 ; Load next byte
12 SBC     #0      ; Subtract 0 + Carry
13 STORE   A, 0x73 ; Store result in second byte
```

12.3.2 Subtraction on the 6502

The 6502 family only supports subtraction with carry with the **SBC**[5] instruction. It does not have a subtract without carry instruction, but you can emulate it by preceding the **SBC**[6] instruction with **SEC**[7] (Set Carry).

5: **SBC** Section 16.63 on page 206

6: **SBC** Section 16.63 on page 206

7: **SEC** Section 16.64 on page 208

For example, subtracting 3 from 7 for a single byte for the 6502:

```
; 6502 code
SEC          ; Set Carry
LDA #7       ; Load 7
SBC #3       ; Subtract 3
STA 0x70     ; Store result
```

Similarly, for multiple byte subtraction:

```
; 6502 code
SEC          ; Set Carry
LDA 0x70     ; Load first byte
SBC #7       ; Subtract 7
STA 0x70     ; Store first byte
LDA 0x71     ; Load second byte
SBC #0       ; Subtract 0 + Carry from first byte
STA 0x71     ; Store second byte
```

Note we use **SEC** before the first subtraction to emulate SUB but not for the subsequent ones.

12.3.3 Subtraction on the 65816

The 16-bit version of the 6502 supports 16-bit subtraction with carry when it's running in native mode, so the following would subtract 7 from the 16-bit value at 0x70 and 0x71:

```
; 6502 code
SEC          ; Set Carry
LDA 0x70     ; Load 16-bit value from 0x70
SBC #7       ; Subtract 7
STA 0x70     ; Store 16-bit result
```

12.3.4 Subtraction on the Z80

The Z80 supports both **SUB**[8] (subtraction without Carry) and **SBC**[9] (subtraction with carry) instructions.

```
; Z80 code
LD  A, (0x70)   ; Load 7-bit value from 0x70
SUB A, #7       ; Add 7 without carry
LD (0x70), A    ; Store result
```

The Z80 also supports 16-bit subtraction with carry when using the 16-bit register pairs with the **SBC** instruction.

```
; Z80 code
LD  HL,(0x70)   ; Load 16-bit value from 0x70
LD  BC, #7      ; Set BC to 7
SBC HL, BC      ; Subtract BC with carry
LD  (0x70), HL  ; Store 16-bit result
```

The Z80 does not have 16-bit register version of the **SUB** instruction.

8: **SUB** Section 22.66 on page 388

9: **SBC** Section 22.60 on page 375

12.4 Comparisons

One of the fundamental operations for any microcomputer is to compare two values be it at a numeric level or at the bit level.

12.4.1 Numeric comparison

At the numeric level a microprocessor will compare two values by subtracting one from the other, then set certain flags based on the result. The result is not stored[10] so is not accessible, only the flags are affected.

10: If the result is required then use the relevant instructions

For example, if we need to compare VALUE1 against VALUE2 then the processor will subtract without carry VALUE2 from VALUE1 and then set the processor flags based on the result.

The flags affected are specific to each processor but this is the general idea

Usually, the Zero Flag will be set if $V_1 - V_2 = 0$ and the Negative Flag set if $V_1 - V_2 < 0$.

Other flags can be set depending on the individual processor.

12.4.2 Binary comparison

Binary comparisons occur when a logical test instruction is executed. These instructions are usually used to test to see if specific bits are set in a value, but can also include the relevant boolean logic instructions AND, OR, Exclusive-OR, and Complement.

The instructions available varies by processor and their names also vary:

6502	BIT, TRB, TSB
Z80	BIT

12.5 Multiplication

As mentioned earlier, most microprocessors do not have a general integer multiplication instruction, leaving that to software. Now this might for a novice programmer seem to be complicated, but it is relatively simple once you figure out how multiplication actually works.

There are different methods taught in schools on how to do multiplication long hand but the way we can write a multiplication routine uses exactly the same methods.

The simplest one of these is where we have two numbers, and we go right to left for each digit of the second number and multiply the first number by that digit and add the totals together.

Figure 12.1 on page 115 shows long multiplication of:

$$1234 \times 1212 = 1495608$$

Figure 12.1: Decimal Multiplication

```
   1234
 × 1212
------
   2468
  +12340
 +246800
+1234000
-------
 1495608
```

Now long multiplication in binary is exactly the same. In fact, it's slightly easier as we only have two digits, 0 and 1. Because of this we don't need to multiply the top operand, just add it if we have a 1 otherwise do nothing for that step.

Figure 12.2 shows the binary multiplication of:

$$5 \times 10 = 50$$

Figure 12.2: Binary Multiplication

```
   101
 × 1010
------
   000
  +1010
 +00000
+101000
------
110010
```

So for the computer to perform the multiplication:

1. Clear the result
2. If the bottom operand is zero we stop as we now have the result.
3. Perform a right shift of the bottom operand so bit-0 goes into the carry flag.
4. If the carry flag is set then we add the top operand to the result.
5. Left shift the top operand
6. Jump back to step 2

12.5.1 Multiplication on the 6502

Listing 12.1 on the following page shows how to multiply two 16-bit numbers on the 6502.

Listing 12.1: 16-bit integer multiplication on the 6502

```
1  VAL1        EQU $80         ; Top operand
2  VAL2        EQU $82         ; Bottom operand
3  RESULT      EQU $84         ; Result
4
5  .MULT       LDA #0          ; Clear the result
6              STA RESULT
7              STA RESULT+1
8  .MULT1      LDA VAL1        ; Check to see if we have zero for VAL1
9              ORA VAL1+1
10             BEQ DONE        ; We have zero so finish
11             LSR VAR1+1      ; Get the right bit by shifting right
12             ROR VAR1        ; for both bytes of VAR1
13             BCC MULT2       ; Skip the add if the bit was 0
14             CLC             ; Add VAL2 to RESULT
15             LDA RESULT
16             ADC VAL2
17             STA RESULT
18             LDA RESULT+1
19             ADC VAL2+1
20             STA RESULT+1
21 .MULT2      ASL VAL2        ; Now shift VAL2 left ready for the next
22             ROL VAL2+1      ; loop iteration
23             JMP MULT1       ; Jump
24 .DONE       RTS             ; Exit
```

Optimisations

For the 65C02 or later, one byte could be saved by replacing the **JMP**[11] on line 23 with **BRA**[12].

Two more bytes could be saved by removing line 5 and using **STZ**[13] instead of **STA**[14] on lines 6 and 7.

On the 65816, it could be rewritten to use 16-bit operations if it's run in native mode and 16-bit accumulators.

11: **JMP** Section 16.31 on page 170
12: **BRA** Section 16.11 on page 146
13: **STZ** Section 16.72 on page 216
14: **STA** Section 16.68 on page 212

12.6 Division

Like Multiplication, most microprocessors leave division to software and similar to Multiplication, this is done as a series of shifts and subtractions.

Figure 12.3 on the next page shows the binary division of:

$$\frac{61}{5} = 12 \textit{ remainder } 1$$

So for the computer to perform the division:

1. Clear the result
2. If the bottom operand is zero we stop as we now have the result.
3. Perform a right shift of the bottom operand so bit-0 goes into the carry flag.
4. If the carry flag is set then we add the top operand to the result.
5. Left shift the top operand
6. Jump back to step 2

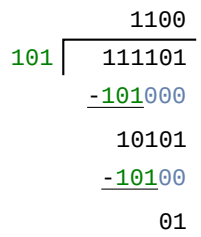


Figure 12.3: Binary Division

Page intentionally blank

The 6502 Microprocessor

Page intentionally blank

13 Vectors

On the 6502, there are a set of vectors located at the top of the address space[1] which point to the interrupt handlers when certain events occur with the processor. As the top of the address space is always in ROM these vectors usually point to addresses within the ROM, or more specifically either the operating system or the monitor.

1: Top of Bank `0x00` on the 65816

13.1 Vectors on the 6502 and 65C02

The original 8-bit processors use just three vectors located in the top 6 bytes of the address space. Table 13.1 on page 122 shows these vectors.

13.1.1 Reset

The most important vector is RESET which defines the start address when the processor first powers up or the reset line is triggered. When the processor is reset, it takes the 16-bit address from the RESET vector, places it into the Program Counter and then starts executing code from that address.

13.1.2 IRQ

The IRQ vector is the main routine which handles interrupts. These are either from the IRQ line or from a software interrupt caused by the **BRK**[2] instruction being executed.

2: **BRK** Section 16.12 on page 147

Hardware uses for interrupts are for hardware requesting the processor to handle incoming data, timer's triggering or the display being refreshed.

However, they can be disabled for short periods of time with I flag using the **CLI**[3] and **SEI**[4] instructions. Games did this for when they needed to perform updates as fast as possible - but they cannot be disabled for long otherwise information could be lost if they are not processed on time.

3: **CLI** Section 16.18 on page 155

4: **SEI** Section 16.66 on page 210

13.1.3 NMI

Non-Maskable Interrupts (NMI) are a higher level of interrupt which gets its own hardware line. Unlike IRQ, these cannot be masked with the I flag, so the processor will always process these events when they happen.

NMI devices are usually reserved for critical devices which must be processed in real-time.

Table 13.1: 6502 Vectors

Address	Vector	Purpose
`0xFFFA`	NMI	Non-Maskable Interrupt handler
`0xFFFC`	RESET	Reset Vector
`0xFFFE`	IRQ	Interrupt Handler

Table 13.2: 65816 Vectors

Emulation mode, e=1		Native mode, e=0		Purpose
Address	Vector	Address	Vector	
`0x00FFF4`	COP	`0x00FFE4`	COP	Coprocessor handler
		`0x00FFE6`	BRK	BRK handler
`0x00FFF8`	ABORT	`0x00FFE8`	ABORT	
`0x00FFFA`	NMI	`0x00FFEA`	NMI	NMI handler
`0x00FFFC`	RESET			Reset Vector
`0x00FFFE`	IRQ / BRK	`0x00FFEE`	IRQ	Interrupt Handler

13.2 Vectors on the 65816

The 16-bit 65816 has the same vectors as the 6502, although it also has additional vectors which are used with both native and emulation modes. As the processor has a 24-bit address bus, all of these vectors are at the top of bank `0x00`.

As you can see in Table 13.2, there are more vectors than just the three used by the 6502 and 65C02.

13.2.1 Reset

The RESET vector is identical to the 6502. In fact, there is no native version of this vector as the processor always runs in emulation mode when it is reset.

13.2.2 IRQ

When in emulation mode the processor uses the same vector used by the 6502 for both IRQ and BRK instructions.

However, when in native mode then both Hardware IRQ and Software BRK get their own vectors. IRQ at `0x00FFEE` and BRK at `0x00FFE6`.

13.2.3 COP

5: **COP** Section 16.21 on page 158

The **COP**[5] instruction is a form of software interrupt like BRK when not used with a hardware coprocessor, so it has its own pair of vectors - one each for native and emulation mode.

13.2.4 ABORT

The ABORT vector is used with the ABORT line which is present on the 65816 and provides a third level of hardware interrupt.

14 Registers

Unlike most microprocessors, the 6502 has relatively few registers available and on the 8-bit processors, with the exception of the PC register they are all only 8-bits in size.

14.1 6502 and 65C02

The 8-bit processors only has six registers, five of them 8-bit.

14.1.1 A Accumulator

The A or Accumulator is a general purpose register used for all arithmetic operations. Operations like addition, subtraction or bit shifts are performed with this register.

14.1.2 X and Y index registers

The X and Y registers are used for referencing memory relative from a starting point. For example if you have a block of memory from `0x1200`, then using the X register you can address any address from `0x1200` ...`0x12FF` using an instruction like `LDA 0x1200,X`. This is why they are called index registers. Most instructions will take the same set of address modes with either register, although there are some exceptions to this rule.

14.1.3 PC Program Counter

The PC or Program Counter is the only 16-bit register available and holds the address of the instruction being

executed. Once an instruction has executed, the PC register is incremented by the size of the instruction so that it then points to the next instruction to execute.

14.1.4 S Stack register

The stack register holds the current position of the top of the processor stack. As the stack is restricted to page `0x01` this is only an 8-bit register. Also, the stack, when empty, points to `0x01FF` as the stack pointer is decremented whenever a byte is pushed to the stack.

14.1.5 P Processor Flags

The P register holds the processor flags. This is described in full in Chapter 15 on page 129.

14.2 65816

The 65816 has the same set of registers as the 8-bit 6502 with a couple of additions. When in emulation mode then the registers are the same as the 6502. However, in native mode then there are some important differences.

14.2.1 P Processor Flags

In native mode some of the bits in the P register differ. These are described in full in Chapter 15 on page 129.

14.2.2 A Accumulator

The Accumulator can be either 8 or 16 bits in size, dependent on the state of the M bit in the P register. When m is set then the Accumulator is 8-bits in size like the 6502. When m is clear, then the Accumulator is 16-bits in size.

14.2.3 B Accumulator

There is a hidden B accumulator which can be accessed in certain conditions. The B accumulator is effectively the upper 8-bits of the 16-bit accumulator.

14.2.4 C Accumulator

Various documentation refer to a C accumulator. This is actually an alternate name for the 16-bit A accumulator, leaving the original A name for the lower 8-bits.

14.2.5 X and Y index registers

In Native mode, the X and Y index registers can be either 8-bit or 16-bit. This is determined with the new X bit in the P register. When X is set then both of the index registers are 8-bit. When X is clear then they are 16-bit.

This does mean that when used against an address then the range is now the full 64K range - although it is restricted to the single 64K memory bank.

14.2.6 S Stack register

TODO

14.2.7 DP Direct Page

TODO

14.2.8 Data Bank register

TODO

14.2.9 Program Bank register

TODO

15 Status Register

With the 6502 family, the flags are stored in the 8-bit status register P.

The flags available depend on if the processor is 8-bit or the 16-bit 65816 in native mode.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

All processors

c carry

d decimal mode

i IRQ interrupt disable

n negative result

v overflow

z zero result

8-bit and 65816 in emulation mode

b BRK caused interrupt

65816 in 16-bit native mode

m 8-bit memory/accumulator

x 8-bit index registers

65816 in either mode

e Emulation mode, not accessible other than via the **XCE**[1] instruction

1: **XCE** Section 16.90 on page 237

15.1 Common to all processors

15.1.1 C Carry

The carry flag is used to carry over a bit between operations, be it addition, subtraction or the shift instructions.

15.1.2 D Decimal Mode

The decimal flag determines if the processor is running in normal or BCD mode.

15.1.3 I Interrupt Disable

The interrupt disable flag allows hardware interrupts to be ignored for a short period of time.

15.1.4 N Negative

The Negative flag represents a result which is negative in two's complement. This is usually when the most significant bit of the result is set.

15.1.5 V Overflow

The overflow flag is set when an operation overflows, or on older processors an external line is used. There is only an instruction to clear this flag. It cannot be set by normal code.

15.1.6 Z Zero

The Zero flag is set when the result is zero. This is usually the case when two values are equal, so operations like **BEQ**[2] operate when the flag is set.

2: **BEQ** Section 16.6 on page 140

15.2 Specific to 8-bit processors

15.2.1 B Break

The Break flag is set when the **BRK**[3] instruction executes. It is used to differentiate against a hardware interrupt which uses the same vector on the 8-bit processors. It is only set in the copy of the flags pushed on the stack when the **BRK** instruction is executed.

3: **BRK** Section 16.12 on page 147

15.3 Specific to 16-bit processors

15.3.1 M Memory and Accumulator mode

On the 65816 when running in native mode, this flag determines if memory accesses or the accumulator are either 8-bit or 16-bit in size.

15.3.2 X Index Register mode

On the 65816 when running in native mode, this flag determines if the index registers are either 8-bit or 16-bit in size.

15.3.3 E Emulation

This flag determines if the 65816 processor is running in native or 6502-emulation mode. You can only access this flag via the **XCE**[4] instruction.

4: **XCE** Section 16.90 on page 237

Page intentionally blank

6502 Instruction Set 16

This chapter describes every single instruction implemented by the 6502 family of microprocessors.

Each section describes an individual instruction, listing the address modes supported, which processors have that instruction and the processor flags that are changed by it, or affects the operation of the instruction.

The following shows the processor flags and the colour scheme used to show how they are affected.

Legend as used in Chapter 16 on page 133

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

- **c** Modified by the instruction
- ***c*** Affects but not modified
- c Not used
- Undefined

16.1 ADC
Add with Carry

Function $A \leftarrow (A) + DATA + C$

Description Add the contents of memory or a literal value, adding one more if the carry bit is setm and store the result in the accumulator.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n set if most significant bit of result is set

v set if signed overflow

z result is zero

c set if unsigned overflow

See also **SBC**[1]

1: **SBC** Section 16.63 on page 206

Hex	Opcode	Length	6502	65C02	65816	Cycles
61	ADC (dp,X)	2	⋆	⋆	⋆	6
63	ADC sr,S	2			⋆	5
65	ADC dp	2	⋆	⋆	⋆	3
67	ADC [dp]	2			⋆	6
69	ADC #	2	⋆	⋆	⋆	2
6D	ADC addr	3	⋆	⋆	⋆	4
6F	ADC (long)	4			⋆	5
71	ADC (dp),Y	2	⋆	⋆	⋆	5
72	ADC (dp)	2		⋆	⋆	5
73	ADC (sr,S),Y	2			⋆	7
75	ADC dp,X	2	⋆	⋆	⋆	4
77	ADC [dp],Y	2			⋆	6
79	ADC addr,Y	3	⋆	⋆	⋆	4
7D	ADC addr,X	3	⋆	⋆	⋆	4
7F	ADC long,X	4			⋆	5

Decimal mode

ADC is affected by the decimal flag so will either operate in binary or decimal mode dependent on that flag.

ADD Add without Carry

As there is no ADD instruction, if you want to add without the carry, you need to use the **CLC**[2] instruction prior to using **ADC**.

2: CLC Section 16.16 on page 153

16.2 AND
Bitwise AND A with Memory

Function $A \leftarrow (A) \wedge DATA$

Description AND performs a logical AND of the value in the accumulator with that of the memory location with the result stored in the accumulator.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n set if most significant bit of result is set

z result is zero

See also **EOR**[3], **ORA**[4]

Figure 16.1: AND truth table

Memory \ Accumulator	0	1
0	0	0
1	0	1

3: **EOR** Section 16.27 on page 165

4: **ORA** Section 16.40 on page 181

Hex	Opcode	Length	6502	65C02	65816	Cycles
21	`AND (dp,X)`	2	⋆	⋆	⋆	6
23	`AND sr,S`	2			⋆	4
25	`AND dp`	2	⋆	⋆	⋆	3
27	`AND [dp]`	2			⋆	6
29	`AND #`	2	⋆	⋆	⋆	2
2D	`AND addr`	3	⋆	⋆	⋆	4
2F	`AND long`	4			⋆	5
31	`AND (dp),Y`	2	⋆	⋆	⋆	5
32	`AND (dp)`	2		⋆	⋆	5
33	`AND (sr,S),Y`	2			⋆	7
35	`AND dp,X`	2	⋆	⋆	⋆	4
37	`AND [dp],Y`	2			⋆	6
39	`AND addr,Y`	3	⋆	⋆	⋆	4
3D	`AND addr,X`	3	⋆	⋆	⋆	4
3F	`AND long,X`	4			⋆	5

16.3 ASL
Shift Memory or Accumulator Left

Function

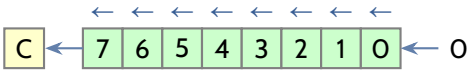


Figure 16.2: Shift left

Description Move the contents of the accumulator or the memory location left by 1 bit position

Bit-0 is cleared

The original Bit-7 falls into the Carry flag.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n set if most significant bit of result is set

z result is zero

c set to the original Bit-7

See also **LSR**[5], **ROL**[6], **ROR**[7]

Hex	Opcode	Length	6502	65C02	65816	Cycles
06	ASL dp	2	⋆	⋆	⋆	5
0A	ASL A	1	⋆	⋆	⋆	2
0E	ASL addr	3	⋆	⋆	⋆	6
16	ASL dp,X	2	⋆	⋆	⋆	6
1E	ASL addr,X	3	⋆	⋆	⋆	7

5: **LSR** Section 16.36 on page 177

6: **ROL** Section 16.58 on page 200

7: **ROR** Section 16.59 on page 201

16.4 BCC
Branch if Carry Clear

Function Go to specified address if $P_C = 0$

Description Branch if the carry flag is clear

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

c Branch if clear.

See also BCS[8]

8: **BCS** Section 16.5 on page 139

The displacement is added to the address of the first instruction following this one. This results in an effective displacement of -126...+129 bytes.

Hex	Opcode	Length	6502	65C02	65816	Cycles
90	BCC near	2	⋆	⋆	⋆	2

16.5 BCS Branch if Carry Set

Function Go to specified address if $P_C = 1$

Description Branch if the carry flag is set

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	**c**	8-bit
n	v	m	x	d	i	z	**c**	16-bit

c Branch if set.

See also **BCC**[9]

9: **BCC** Section 16.4 on page 138

The displacement is added to the address of the first instruction following this one. This results in an effective displacement of -126…+129 bytes.

Hex	Opcode	Length	6502	65C02	65816	Cycles
`B0`	`BCS near`	2	⋆	⋆	⋆	2

16.6 BEQ
Branch if Equal

Function Go to specified address if $P_Z = 1$

Description Branch if the last result was zero

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

z Branch if set.

See also **BNE**[10]

10: **BNE** Section 16.9 on page 144

The displacement is added to the address of the first instruction following this one. This results in an effective displacement of -126...+129 bytes.

Hex	Opcode	Length	6502	65C02	65816	Cycles
F0	BEQ near	2	⋆	⋆	⋆	2

16.7 BIT
Test Memory Bits

Function

$$P_z \leftarrow \overline{(A) \wedge (M)}$$
$$P_n \leftarrow (M_7)$$
$$P_v \leftarrow (M_6)$$

Description Perform a logical AND of the memory and accumulator, then set the Z flag if this result is zero.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n set to the highest bit of memory

v set to next to highest bit of memory

z set if (memory AND A) == 0

See also **TRB**[11], **TSB**[12]

11: **TRB** Section 16.78 on page 223

12: **TSB** Section 16.79 on page 225

Hex	Opcode	Length	6502	65C02	65816	Cycles
24	`BIT dp`	2	⋆	⋆	⋆	3
2C	`BIT addr`	3	⋆	⋆	⋆	4
34	`BIT dp,X`	2		⋆	⋆	4
3C	`BIT addr,X`	3		⋆	⋆	4
89	`BIT #`	2		⋆	⋆	2

The **BIT** instruction is usually followed by one of the conditional branch instructions.

Immediate address mode

The N and V flags are not modified when Immediate address mode is in use

All other address modes

For all address modes other than Immediate, copy bits 7 and 6 of the memory location[13] to the N and V flags.

Modifying memory

The memory location and accumulator are not modified by this instruction. To modify memory at the same time, then the **TRB**[14] and **TSB**[15] instructions should be used.

13: On the 65816 in 16-bit accumulator/memory mode (m=0) then bits 15 and 14 are used

14: **TRB** Section 16.78 on page 223

15: **TSB** Section 16.79 on page 225

16.8 BMI Branch if Minus

Function Go to specified address if $P_N = 1$

Description Branch if the most significant bit of the last result was set

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Branch if set.

See also **BPL**[16]

16: **BPL** Section 16.10 on page 145

The displacement is added to the address of the first instruction following this one. This results in an effective displacement of -126…+129 bytes.

Hex	Opcode	Length	6502	65C02	65816	Cycles
30	`BMI near`	2	⋆	⋆	⋆	2

16.9 BNE
Branch if Not equal

Function Go to specified address if $P_Z = 0$

Description Branch if the last result was not equal to zero

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

z Branch if clear.

See also **BEQ**[17]

17: **BEQ** Section 16.6 on page 140

The displacement is added to the address of the first instruction following this one. This results in an effective displacement of -126...+129 bytes.

Hex	Opcode	Length	6502	65C02	65816	Cycles
D0	BNE near	2	⋆	⋆	⋆	2

16.10 BPL
Branch if Positive

Function Go to specified address if $P_N = 0$

Description Branch if the most significant bit of the last result was clear

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Branch if clear.

See also **BMI**[18]

The displacement is added to the address of the first instruction following this one. This results in an effective displacement of -126...+129 bytes.

18: **BMI** Section 16.8 on page 143

16.11 BRA
Branch Always

Description Branch to the specified address

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

See also **BRL**[19] **JMP**[20]

The displacement is added to the address of the first instruction following this one. This results in an effective displacement of -126...+129 bytes.

Relative jumps outside of this range are only possible with the 65816 in native mode with the **BRL** instruction.

Hex	Opcode	Length	6502	65C02	65816	Cycles
80	BRA near	2		⋆	⋆	3

19: **BRL** Section 16.13 on page 150

20: **JMP** Section 16.31 on page 170

16.12 BRK Software Interrupt

Function Force a software interrupt

Description **BRK** forces a software interrupt and is unaffected by the I interrupt disable flag.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

All processors

i Interrupt disable flag is set, disabling hardware IRQ interrupts.

65C02 and 65816

d Cleared on all processors after the 6502 to enable binary arithmetic.
This flag is not touched on the 6502, so the handler normally has to clear this flag itself.

6502, 65C02 8-bit processors
65816 in 8-bit emulation mode

b set in the copy of the P register pushed on the stack.
The actual register never has this bit set.

See also **RTI**[21], **COP**[22]

21: **RTI** Section 16.60 on page 202
22: **COP** Section 16.21 on page 158

See Subsection 16.12.3 on page 149 on the discrepancy with the **BRK** instruction size.

Hex	Opcode	Length	6502	65C02	65816	Cycles
00	BRK	2	⋆	⋆	⋆	7

16.12.1 6502, 65C02 and emulation mode

When the **BRK** is invoked on the 8-bit processors:

- The program counter is incremented by two, then pushed to the stack
- The status register, with the B break flag set, is pushed onto the stack
- The interrupt disable flag I is set
- The program counter is set to the interrupt vector at `0xFFFE` which should point to the interrupt handler.

As the vector is shared with hardware interrupts, the code pointed to by the interrupt vector needs to handle this:

```
0000 68     PLA         ; Set A to status on
0001 48     PHA         ; stack without changing
     it
0002 2910   AND #0x10   ; Check BRK bit
0004 D0xx   BNE isbrk   ; branch if set
```

16.12.2 65816 native mode

When the **BRK** is invoked on the 65816 running in 16-bit native mode:

- The program counter bank register is pushed onto the stack
- The program counter is incremented by two, then pushed to the stack
- The status register is pushed onto the stack
- The interrupt disable flag I is set
- The program counter bank register is set to zero and the program counter is set to the break vector at `0x00FFE6`.

16.12.3 Instruction size discrepancy

The **BRK** instruction is defined as a single byte instruction, which is how the instruction has been written in all documentation and every assembler for the processor since its introduction.

However, when it is invoked, the Program Counter is incremented by 2. This allows for a one-byte signature value indicating which break caused the interrupt.

Even if the signature byte is not required, it must either be there for the **RTI**[23] instruction which returns control to the caller must manually decrement the return address.

23: **RTI** Section 16.60 on page 202

As this can be tricky, most operating systems require **BRK** to take up 2 bytes in memory.

16.12.4 BBC Micro & Acorn Electron usage

On the BBC Microcomputer[24], the standard implemented by the operating system MOS is for the **BRK** to be followed by an error number and then the error message terminated with a `0`.

24: This also applies to the Acorn Electron and Acorn Atom as they share the same heritage with MOS

Some Paged ROMs will copy the instruction to address `0x0100` then jump to that address as the handler is usually in the active language ROM and would otherwise not have access to it.

This works as the stack runs down from `0x01FF` but also as on this platform, the BRK is never returned from so the stack is reset.

16.13 BRL
Branch Always Long

Description Branch to the specified address

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

See also **BRA**[25] **JMP**[26]

25: **BRA** Section 16.11 on page 146

26: **JMP** Section 16.31 on page 170

BRL is a three-byte instruction taking a relative offset that is valid for the current 64K program bank.

This is in effect, a **JMP** with the advantage of allowing for relative displacements rather than abolute addresses, allowing for relocatable code.

The downside is that **JMP** is one cycle faster than **BRL**.

Check validity of this statement

Hex	Opcode	Length	6502	65C02	65816	Cycles
82	`BRL near`	3			⋆	4

16.14 BVC
Branch if Overflow Clear

Function Go to specified address if $P_V = 0$

Description Branch if the overflow flag is clear

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

v Branch if clear.

See also **BVS**[27]

27: **BVS** Section 16.15 on page 152

The displacement is added to the address of the first instruction following this one. This results in an effective displacement of -126...+129 bytes.

Hex	Opcode	Length	6502	65C02	65816	Cycles
50	BVC near	2	⋆	⋆	⋆	2

16.15 BVS
Branch if Overflow Set

Function Go to specified address if $P_V = 1$

Description Branch if the overflow flag is set

Flags

	7	6	5	4	3	2	1	0	
	n	*v*		b	d	i	z	c	8-bit
	n	*v*	m	x	d	i	z	c	16-bit

v Branch if set.

See also **BVC**[28]

28: **BVC** Section 16.14 on page 151

The displacement is added to the address of the first instruction following this one. This results in an effective displacement of -126...+129 bytes.

Hex	Opcode	Length	6502	65C02	65816	Cycles
`70`	`BVS near`	2	⋆	⋆	⋆	2

16.16 CLC
Clear Carry Flag

Description Clear the carry flag

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

c Cleared

See also **SEC**[29]

29: **SEC** Section 16.64 on page 208

This is often required before an **ADC** instruction to perform addition without carry.

Hex	Opcode	Length	6502	65C02	65816	Cycles
18	CLC	1	⋆	⋆	⋆	2

16.17 CLD
Clear Decimal Flag

Description Clear the decimal flag

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

d Cleared

See also **SED**[30]

30: **SED** Section 16.65 on page 209

Hex	Opcode	Length	6502	65C02	65816	Cycles
D8	CLD	1	⋆	⋆	⋆	2

16.18 CLI
Enable Hardware Interrupts

Description Clear the interrupt flag enabling hardware interrupts

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

i Cleared

See also **SEI**[31]

31: **SEI** Section 16.66 on page 210

As the interrupt flag is set when the interrupt handler is invoked, that handler must re-enable interrupts before returning using this instruction.

Some time-critical code, usually in realtime systems or games, wrap a section of code between **SEI** ...**CLI** instructions, so they get the processor to themselves.

However, whilst the I flag is set, then it is possible that a hardware interrupt will go unnoticed if it's set for too long a period.

Hex	Opcode	Length	6502	65C02	65816	Cycles
58	CLI	1	⋆	⋆	⋆	2

16.19 CLV
Clear Overflow Flag

Description Clear the carry flag

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

v Cleared

Unlike most flag operations, there is no equivalent **SEV** instruction to set the Overflow flag.

It can only be set as the result of certain instructions or, on older processors by a rarely used hardware pin.

Hex	Opcode	Length	6502	65C02	65816	Cycles
B8	CLV	1	⋆	⋆	⋆	2

16.20 CMP Compare Accumulator with Memory

Function Compare A with Memory

Description The **CMP** instruction subtracts the value from the effective address in memory from the accumulator, setting the C, Z and N flags based on the result.

The result itself is not saved.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if must significant bit of result is set, else cleared

z Set if result is zero $A = M$, else cleared $A \neq M$

c Set if no borrow required $A \geq M$, else cleared $A < M$

See also **CPX**[32], **CPY**[33]

Table 16.1 shows how to handle the result from **CMP**.

Result	Op
$A = M$	**BEQ**
$A \neq M$	**BNE**
$A \geq M$	**BCS**
$A < M$	**BCC**

32: **CPX** Section 16.22 on page 160

33: **CPY** Section 16.23 on page 161

Table 16.1: Handling result from **CMP**, **CPX** or **CPY**

16.21 COP CO-Processor

Function Force a software interrupt

Description **COP** causes a software interrupt in a similar manner to **BRK**[34], except it uses its own vector.

COP can also be trapped by a physical coprocessor like a Floating Point Unit or a Graphics processor in order to invoke a function on said coprocessor.

34: **BRK** Section 16.12 on page 147

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

i Set to disable hardware IRQ interrupts.

d Cleared

See also **BRK**[35], **RTI**[36]

35: **BRK** Section 16.12 on page 147

36: **RTI** Section 16.60 on page 202

Hex	Opcode	Length	6502	65C02	65816	Cycles
02	COP const	2			⋆	7

As **COP** is done at the software level, it's not affected by the I interrupt disable flag.

Unlike **BRK**, assemblers must account for the byte after the **COP** instruction.

The value of the signature byte in the range `0x00` ... `0x7F` are available for use with software implemented COP Handlers.

Those in the range `0x80` ...`0xFF` have been reserved for hardware use.

16.21.1 6502 Emulation Mode

If the 65816 is in 6502 emulation mode:

- The program counter is incremented by two and pushed onto the stack
- The status register is pushed onto the stack
- The interrupt disable flag is set
- The program counter is loaded from the emulation mode co-processor vector at `0xFFF4`.
- The D flag is cleared after **COP** has been executed.

16.21.2 65816 Native Mode

- The program counter bank register is pushed onto the stack.
- The program counter is incremented by two and pushed onto the stack
- The status register is pushed onto the stack
- The interrupt disable flag is set
- The program counter bank is set to 0 and the program counter is loaded from the emulation mode co-processor vector at `0x00FFE4`.
- The D flag is cleared after **COP** has been executed.

16.22 CPX
Compare X with Memory

Function Compare X with Memory

Description The **CPX** instruction subtracts the value from the effective address in memory from the X register, setting the C, Z and N flags based on the result.

The result itself is not saved.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if must significant bit of result is set, else cleared

z Set if result is zero $X = M$, else cleared $X \neq M$

c Set if no borrow required $X \geq M$, else cleared $X < M$

See also **CMP**[37], **CPY**[38]

37: **CMP** Section 16.20 on page 157

38: **CPY** Section 16.23 on page 161

Table 16.1 on page 157 shows how to handle the result from **CPX**.

Hex	Opcode	Length	6502	65C02	65816	Cycles
E0	CPX #	2	⋆	⋆	⋆	2
E4	CPX dp	2	⋆	⋆	⋆	3
EC	CPX addr	3	⋆	⋆	⋆	4

16.23 CPY
Compare Y with Memory

Function Compare Y with Memory

Description The **CPY** instruction subtracts the value from the effective address in memory from the Y register, setting the C, Z and N flags based on the result.

The result itself is not saved.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if must significant bit of result is set, else cleared

z Set if result is zero $Y = M$, else cleared $Y \neq M$

c Set if no borrow required $Y \geq M$, else cleared $Y < M$

See also **CMP**[39], **CPX**[40]

Table 16.1 on page 157 shows how to handle the result from **CPY**.

Hex	Opcode	Length	6502	65C02	65816	Cycles
C0	CPY #	2	⋆	⋆	⋆	2
C4	CPY dp	2	⋆	⋆	⋆	3
CC	CPY addr	3	⋆	⋆	⋆	4

39: **CMP** Section 16.20 on page 157

40: **CPX** Section 16.22 on page 160

16.24 DEC
Decrement by one A or Memory

Function $M \leftarrow (M) - 1$

Description The contents of the specified memory address is decremented by 1.

The result is stored back at the specified memory address.

Note The version with opcode `0x3A` uses the Accumulator as the target to be decremented instead of a memory address.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if must significant bit of result is set, else cleared

z Set if result is zero, else cleared

See also **DEX**[41], **DEY**[42], **INC**[43], **INX**[44], **INY**[45]

Hex	Opcode	Length	6502	65C02	65816	Cycles
`3A`	`DEC A`	1	⋆	⋆	⋆	2
`C6`	`DEC dp`	2	⋆	⋆	⋆	5
`CE`	`DEC addr`	3	⋆	⋆	⋆	6
`D6`	`DEC dp,X`	2	⋆	⋆	⋆	6
`DE`	`DEC addr,X`	3	⋆	⋆	⋆	7

41: **DEX** Section 16.25 on page 163
42: **DEY** Section 16.26 on page 164
43: **INC** Section 16.28 on page 167
44: **INX** Section 16.29 on page 168
45: **INY** Section 16.30 on page 169

16.25 DEX
Decrement X by one

Function $X \leftarrow (X) - 1$

Description Decrement the X register by 1 storing the result back in the X register.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if must significant bit of result is set, else cleared

z Set if result is zero, else cleared

See also **DEC**[46], **DEY**[47], **INC**[48], **INX**[49], **INY**[50]

Hex	Opcode	Length	6502	65C02	65816	Cycles
CA	DEX	1	⋆	⋆	⋆	2

46: **DEC** Section 16.24 on page 162

47: **DEY** Section 16.26 on page 164

48: **INC** Section 16.28 on page 167

49: **INX** Section 16.29 on page 168

50: **INY** Section 16.30 on page 169

16.26 DEY
Decrement Y by one

Function $Y \leftarrow (Y) - 1$

Description Decrement the Y register by 1 storing the result back in the Y register.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if must significant bit of result is set, else cleared

z Set if result is zero, else cleared

See also **DEC**[51], **DEX**[52], **INC**[53], **INX**[54], **INY**[55]

Hex	Opcode	Length	6502	65C02	65816	Cycles
88	DEY	1	⋆	⋆	⋆	2

51: **DEC** Section 16.24 on page 162

52: **DEX** Section 16.25 on page 163

53: **INC** Section 16.28 on page 167

54: **INX** Section 16.29 on page 168

55: **INY** Section 16.30 on page 169

16.27 EOR Exclusive OR

Function $A \leftarrow (A)\ (M)$

Description **EOR** performs a bitwise logical Exclusive-OR of the value in the accumulator with that of the memory location with the result stored in the accumulator.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if must significant bit of result is set, else cleared

z Set if result is zero, else cleared

See also **AND**[56], **ORA**[57]

Figure 16.3: EOR truth table

Memory \ Accumulator	0	1
0	0	1
1	1	0

56: **AND** Section 16.2 on page 136

57: **ORA** Section 16.40 on page 181

Hex	Opcode	Length	6502	65C02	65816	Cycles
41	EOR (dp,X)	2	⋆	⋆	⋆	6
43	EOR sr,S	2			⋆	4
45	EOR dp	2	⋆	⋆	⋆	3
47	EOR [dp]	2			⋆	6
49	EOR #	2	⋆	⋆	⋆	2
4D	EOR addr	3	⋆	⋆	⋆	4
4F	EOR long	4			⋆	5
51	EOR (dp),Y	2	⋆	⋆	⋆	5
52	EOR (dp)	2		⋆	⋆	5
53	EOR (sr,S),Y	2			⋆	7
55	EOR dp,X	2	⋆	⋆	⋆	4
57	EOR [dp],Y	2			⋆	6
59	EOR addr,Y	3	⋆	⋆	⋆	4
5D	EOR addr,X	3	⋆	⋆	⋆	4
5F	EOR long,X	4			⋆	5

16.27.1 Examples

A common use of **EOR** is to convert characters between Upper and Lower case.

In ASCII[58] characters 0x41…0x5A represent the latin alphabet in uppercase whilst characters 0x61…0x7A represent lower case.

As the difference between the two sets is just a single bit, applying **EOR** with 0x20 will, for characters in those ranges, change case.

58: Section **??** on page ?? has an ASCII Table

16.28 INC
Increment by one A or Memory

Function $M \leftarrow (M) + 1$

Description The contents of the specified memory address is incremented by 1.

The result is stored back at the specified memory address.

Note The version with opcode `0x1A` uses the Accumulator as the target to be incremented instead of a memory address.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if must significant bit of result is set, else cleared

z Set if result is zero, else cleared

See also DEC[59], DEX[60], DEY[61], INX[62], INY[63]

Hex	Opcode	Length	6502	65C02	65816	Cycles
`1A`	`INC A`	1	⋆	⋆	⋆	2
`E6`	`INC dp`	2	⋆	⋆	⋆	5
`EE`	`INC addr`	3	⋆	⋆	⋆	6
`F6`	`INC dp,X`	2	⋆	⋆	⋆	6
`FE`	`INC addr,X`	3	⋆	⋆	⋆	7

59: DEC Section 16.24 on page 162
60: DEX Section 16.25 on page 163
61: DEY Section 16.26 on page 164
62: INX Section 16.29 on page 168
63: INY Section 16.30 on page 169

16.29 INX
Increment X by one

Function $X \leftarrow (X) + 1$

Description Increment the X register by 1 storing the result back in the X register.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if must significant bit of result is set, else cleared

z Set if result is zero, else cleared

See also **DEC**[64], **DEC**[65], **DEY**[66], **INC**[67], **INY**[68]

Hex	Opcode	Length	6502	65C02	65816	Cycles
E8	INX	1	⋆	⋆	⋆	2

64: **DEC** Section 16.24 on page 162

65: **DEC** Section 16.24 on page 162

66: **DEY** Section 16.26 on page 164

67: **INC** Section 16.28 on page 167

68: **INY** Section 16.30 on page 169

16.30 INY
Increment Y by one

Function $Y \leftarrow (Y) + 1$

Description Increment the Y register by 1 storing the result back in the Y register.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if must significant bit of result is set, else cleared

z Set if result is zero, else cleared

See also **DEC**[69], **DEC**[70], **DEY**[71], **INC**[72], **INX**[73]

Hex	Opcode	Length	6502	65C02	65816	Cycles
C8	INY	1	⋆	⋆	⋆	2

69: **DEC** Section 16.24 on page 162

70: **DEC** Section 16.24 on page 162

71: **DEY** Section 16.26 on page 164

72: **INC** Section 16.28 on page 167

73: **INX** Section 16.29 on page 168

16.31 JMP
Jump to address

Function *PC* ← *ADDRESS*

Description Take a 16-bit destination address from memory and place it into the Program Counter, causing a jump to occur.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

For the 24-bit destination addresses, the Program Counter is loaded with the 16-bit address as before but the third byte of the address is also placed into the Program Counter Bank register.

Hex	Opcode	Length	6502	65C02	65816	Cycles
4C	`JMP addr`	3	⋆	⋆	⋆	4
5C	`JMP long`	4			⋆	4
6C	`JMP (addr)`	3	⋆	⋆	⋆	5
7C	`JMP (addr,X)`	3		⋆	⋆	6
DC	`JMP [long]`	3			⋆	6

16.31.1 JML alternate mnemonic

To avoid confusion between the absolute and long address instructions, some assemblers use **JML** as the instruction name for the absolute long instructions.

16.31.2 JMP (addr) on the 65C02

On the 65C02, the **JMP** (addr) (0x6C) instruction takes an additional cycle to complete, compared to the other processors.

16.31.3 JMP (addr) bug on the 6502

On the 6502, if the low byte of the address is 0xFF then when the processor reads the second byte of the address, it reads it from the wrong location.

For example, if the instruction is JMP (0x02FF), then it should get the low byte from 0x02FF and the high byte from 0x0300. However it doesn't increment the page when it rolls over so instead it gets the high byte from 0x0200 which is incorrect.

16.32 JSR
Jump to subroutine

Function

$$STACK \leftarrow \begin{cases} (PC)+2 & \text{16-bit address} \\ (PCB) & \text{24-bit address} \\ (PC)+3 \end{cases}$$

$$PC \leftarrow ADDRESS$$

Description Transfer control to the subroutine at the location specified, after first pushing onto the stack the return address.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

See also **RTS**[74], **RTL**[75]

74: **RTS** Section 16.62 on page 205

75: **RTL** Section 16.61 on page 204

Hex	Opcode	Length	6502	65C02	65816	Cycles
20	`JSR addr`	3	⋆	⋆	⋆	6
22	`JSR long`	4			⋆	8
FC	`JSR (addr,X)`	3			⋆	8

16.32.1 JSL alternate mnemonic

Some assemblers use JSL for the **JSR** long instruction to differentiate between the 16-bit and 24-bit versions.

16.32.2 JSR with 16-bit addresses

For 16-bit addresses the address pushed onto the stack will be the current Program Counter +2.

Subroutines called via a 16-bit address must use the **RTS** instruction as it expects only two bytes to be present on the stack for the return address.

16.32.3 JSR long

For 24-bit addresses, the Program Counter Bank is pushed to the stack followed by the Program Counter as it would have been if it was a 16-bit address.

Subroutines called via a long address mode must return using the **RTL** instruction as it expects three bytes to be present on the stack for the return address.

16.33 LDA
Load Accumulator from Memory

Function $A \leftarrow DATA$

Description Load the accumulator with data from the specified address.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if the most significant bit of the result is set, else cleared

z Set if the result is zero, else cleared.

See also **LDX**[76], **LDY**[77]

76: **LDX** Section 16.34 on page 175

77: **LDY** Section 16.35 on page 176

Hex	Opcode	Length	6502	65C02	65816	Cycles
A1	LDA (dp,X)	2	⋆	⋆	⋆	6
A3	LDA sr,S	2			⋆	4
A5	LDA dp	2	⋆	⋆	⋆	3
A7	LDA [dp]	2			⋆	6
A9	LDA #	2	⋆	⋆	⋆	2
AD	LDA addr	3	⋆	⋆	⋆	4
AF	LDA long	3			⋆	4
B1	LDA (dp),Y	2	⋆	⋆	⋆	5
B2	LDA (dp)	2		⋆	⋆	5
B3	LDA (sr,S),Y	2			⋆	7
B5	LDA dp,X	2	⋆	⋆	⋆	4
B7	LDA [dp],Y	2			⋆	6
B9	LDA addr,Y	3	⋆	⋆	⋆	4
BD	LDA addr,X	3	⋆	⋆	⋆	4
BF	LDA long,X	4			⋆	5

16.34 LDX
Load X from Memory

Function $X \leftarrow DATA$

Description Load the X index register with data from the specified address.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if the most significant bit of the result is set, else cleared

z Set if the result is zero, else cleared.

See also **LDA**[78], **LDY**[79]

78: **LDA** Section 16.33 on page 174

79: **LDY** Section 16.35 on page 176

Hex	Opcode	Length	6502	65C02	65816	Cycles
A2	LDX #	2	⋆	⋆	⋆	2
A6	LDX dp	2	⋆	⋆	⋆	3
AE	LDX addr	3	⋆	⋆	⋆	4
B6	LDX dp,Y	2	⋆	⋆	⋆	4
BE	LDX addr,Y	3	⋆	⋆	⋆	4

16.35 LDY
Load Y from Memory

Function $Y \leftarrow DATA$

Description Load the Y index register with data from the specified address.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if the most significant bit of the result is set, else cleared

z Set if the result is zero, else cleared.

See also **LDA**[80], **LDX**[81]

80: **LDA** Section 16.33 on page 174

81: **LDX** Section 16.34 on page 175

Hex	Opcode	Length	6502	65C02	65816	Cycles
A0	LDY #	2	⋆	⋆	⋆	2
A4	LDY dp	2	⋆	⋆	⋆	3
AC	LDY addr	3	⋆	⋆	⋆	4
B4	LDY dp,X	2	⋆	⋆	⋆	4
BC	LDY addr,X	3	⋆	⋆	⋆	4

16.36 LSR
Logical Shift Memory or A Right

Function

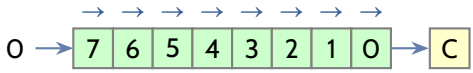


Figure 16.4: Shift right

Description Move the contents of the accumulator or the memory location right by 1 bit position. Bit-7 is cleared. The original bit-0 falls into the Carry flag.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n cleared

z result is zero

c the original Low bit becomes carry

See also ASL[82], ROL[83], ROR[84]

Hex	Opcode	Length	6502	65C02	65816	Cycles
46	`LSR dp`	2	⋆	⋆	⋆	5
4A	`LSR A`	1	⋆	⋆	⋆	2
4E	`LSR addr`	3	⋆	⋆	⋆	6
56	`LSR dp,X`	2	⋆	⋆	⋆	6
5E	`LSR addr,X`	3	⋆	⋆	⋆	7

82: **ASL** Section 16.3 on page 137
83: **ROL** Section 16.58 on page 200
84: **ROR** Section 16.59 on page 201

16.37 MVN
Move block Next

Description Move bytes between blocks

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

Hex	Opcode	Length	6502	65C02	65816	Cycles
54	`MVN src, dest`	3			⋆	n

16.38 MVP
Move block Previous

Description Move bytes between blocks

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

Hex	Opcode	Length	6502	65C02	65816	Cycles
44	`MVP src, dest`	3			⋆	n

16.39 NOP
No Operation

Description A NOP takes no action and does not effect any registers except the program counter.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

NOP's are usually used for timing loops as each NOP takes 2 cycles.

Hex	Opcode	Length	6502	65C02	65816	Cycles
EA	NOP	1	⋆	⋆	⋆	2

16.40 ORA OR Accumulator with Memory

Format $A \leftarrow (A) \vee DATA$

Description ORA performs a bitwise logical OR of the value in the accumulator with that of the memory location with the result stored in the accumulator.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n set if most significant bit of result is set

z result is zero

Figure 16.5: ORA truth table

Memory \ Accumulator	0	1
0	0	1
1	1	1

Hex	Opcode	Length	6502	65C02	65816	Cycles
01	`ORA (dp,X)`	2	⋆	⋆	⋆	6
03	`ORA sr,S`	2			⋆	4
05	`ORA dp`	2	⋆	⋆	⋆	5
07	`ORA [dp]`	2			⋆	6
09	`ORA #`	2	⋆	⋆	⋆	2
0D	`ORA addr`	3	⋆	⋆	⋆	4
0F	`ORA long`	4			⋆	5
11	`ORA (dp),Y`	2		⋆	⋆	5
12	`ORA (dp)`	2			⋆	5
13	`ORA (sr,S),Y`	2			⋆	7
15	`ORA dp,X`	2	⋆	⋆	⋆	4
17	`ORA [dp],Y`	2			⋆	6
19	`ORA addr,Y`	3	⋆	⋆	⋆	4
1D	`ORA addr,X`	3	⋆	⋆	⋆	4
1F	`ORA long,X`	4	⋆	⋆	⋆	5

16.41 PEA
Push Effective Absolute Address

Function *STACK* ← *addr*

Description Push the 16-bit operand onto the stack.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

This instruction always pushes 16-bits of data regardless of the m or x mode select flags.

Although the instruction suggests it pushes an address to the stack, the address can be any 16-bit value that needs to be pushed to the stack.

This could be confusing with the instruction referring to an absolute address, when the address is never accessed. This should really be a 16-bit implied address mode rather than an absolute one.

Hex	Opcode	Length	6502	65C02	65816	Cycles
F4	PEA	3			⋆	5

16.42 PEI
Push Effective Indirect Address

Function $STACK \leftarrow (addr)$

Description Retrieve the 16-bit value located at the direct page address in the operand and push it onto the stack.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

This instruction always pushes 16-bits of data regardless of the m or x mode select flags.

Although the instruction suggests it pushes an address to the stack, the address can be any 16-bit value that needs to be pushed to the stack.

Hex	Opcode	Length	6502	65C02	65816	Cycles
D4	PEI	2			⋆	6

16.43 PER
Push Relative Indirect Address

Format $STACK \leftarrow (PC) + label$

Description Add the current value of the program counter to the 16-bit signed displacement in the instruction and push the result to the stack.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

Hex	Opcode	Length	6502	65C02	65816	Cycles
62	PER	3			⋆	6

This instruction always pushes 16-bits of data regardless of the m or x mode select flags.

Due to the instruction using a displacement relative to the current location of the Program Counter, this instruction is usually used for writing self-relocatable code.

As is the case with the branch instructions, the location being referenced must be within the current program bank.

16.44 PHA
Push Accumulator

Format $STACK \leftarrow (A)$

Description Push the accumulator onto the stack.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

On all processors and the 65816 in 8-bit accumulator mode, a single byte is pushed to the stack.

On the 65816 in 16-bit accumulator mode (m=0), 2 bytes are pushed to the stack, high byte first then the low byte.

Hex	Opcode	Length	6502	65C02	65816	Cycles
48	PHA	1	⋆	⋆	⋆	3

16.45 PHB
Push Data Bank Register

Format $STACK \leftarrow (B)$

Description Push the data bank register onto the stack.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

As the register is only 8-bits in size, only a single byte is pushed to the stack.

Hex	Opcode	Length	6502	65C02	65816	Cycles
8B	PHB	1			⋆	3

16.46 PHD
Push Direct Page Register

Function $STACK \leftarrow (D)$

Description Push the contents of the direct page register D onto the stack.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

As D is always a 16-bit register, this instruction always pushes 2 bytes to the stack, high byte first then the low byte.

Hex	Opcode	Length	6502	65C02	65816	Cycles
0B	PHD	1			⋆	4

16.47 PHK
Push Direct Page Register

Format $STACK \leftarrow (PBR)$

Description Push the program bank register onto the stack.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

As the register is only 8-bits, this instruction always pushes a single byte to the stack.

Hex	Opcode	Length	6502	65C02	65816	Cycles
4B	PHK	1			⋆	3

16.48 PHP Push Status Flags

Format $STACK \leftarrow (P)$

Description Push the contents of the processor status register P to the stack.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

Hex	Opcode	Length	6502	65C02	65816	Cycles
08	PHP	1	⋆	⋆	⋆	4

As this is an 8-bit register, only a single byte is pushed to the stack.

65816

On the 65816, the e bit which indicates 6502 emulation mode, is not pushed onto the stack. The only way to access the e flag is via the **XCE**[85] instruction.

85: **XCE** Section 16.90 on page 237

16.49 PHX
Push Index Register X

Format $STACK \leftarrow (X)$

Description Push the X index register onto the stack.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

Listing 16.1: Implementing **PHX** on the 6502

```
; Saving Y on
; the stack
;
; 65C02 onwards
PHX
;
; 6502
TXA
PHA
```

Hex	Opcode	Length	6502	65C02	65816	Cycles
DA	PHX	1		⋆	⋆	3

On all processors and the 65816 in 8-bit index register mode (x=1), a single byte is pushed to the stack.

On the 65816 in 16-bit index register mode (x=0), 2 bytes are pushed to the stack, high byte first then the low byte.

6502

Before **PHX** was introduced on the 65C02, this was implemented by transferring X to A first then pushing to the stack.The downside to this was it used two bytes and required the accumulator.

16.50 PHY Push Index Register Y

Format $STACK \leftarrow (Y)$

Description Push the Y index register onto the stack.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

Hex	Opcode	Length	6502	65C02	65816	Cycles
5A	PHY	1		⋆	⋆	3

On all processors and the 65816 in 8-bit accumulator mode, a single byte is pushed to the stack.

On the 65816 in 16-bit index register mode (x=0), 2 bytes are pushed to the stack, high byte first then the low byte.

Listing 16.2: Implementing **PHY** on the 6502

```
; Saving Y on
; the stack
;
; 65C02 onwards
PHY
;
; 6502
TYA
PHA
```

6502

Before **PHY** was introduced on the 65C02, this was implemented by transferring Y to A first then pushing to the stackThe downside to this was it used two bytes and required the accumulator.

16.51 PLA
Pull Accumulator

Format $A \leftarrow (STACK)$

Description Pull the accumulator from the stack.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if the most significant bit of the pulled value is set, else cleared

z Set if value pulled is zero, else cleared.

On all processors and the 65816 in 8-bit accumulator mode, a single byte is pushed to the stack.

On the 65816 in 16-bit accumulator mode (m=0), 2 bytes are pushed to the stack, high byte first then the low byte.

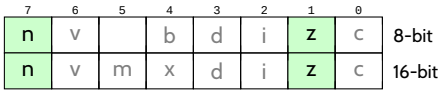
Hex	Opcode	Length	6502	65C02	65816	Cycles
68	PLA	1	⋆	⋆	⋆	3

16.52 PLB
Pull Data Bank Register

Format $PHB \leftarrow (STACK)$

Description Pull the data bank register from the stack.

Flags



n Set if the most significant bit of the pulled value is set, else cleared

z Set if value pulled is zero, else cleared.

As the register is only 8-bits in size, only a single byte is pulled from the stack.

Hex	Opcode	Length	6502	65C02	65816	Cycles
AB	PLB	1			⋆	4

16.53 PLD
Pull Direct Page Register

Format $D \leftarrow (STACK)$

Description Pull the direct page register D from the stack.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if the most significant bit of the pulled value is set, else cleared

z Set if value pulled is zero, else cleared.

As D is always a 16-bit register, this instruction always pulls 2 bytes from the stack, the low byte first, then the high byte.

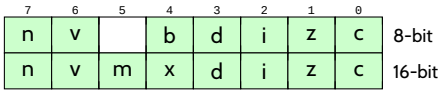
Hex	Opcode	Length	6502	65C02	65816	Cycles
2B	PLD	1			⋆	5

16.54 PLP
Pull Status Flags

Format $D \leftarrow (STACK)$

Description Pull the processor status register P from the stack.

Flags



All processors

c carry

d decimal mode

i IRQ interrupt disable

n negative result

v overflow

z zero result

8-bit & 65816 in emulation mode

b BRK caused interrupt

65816 in 16-bit native mode

m 8-bit memory/accumulator

x 8-bit index registers

65816 in either mode

e Emulation mode, not accessible other than via the **XCE**[86] instruction

86: **XCE** Section 16.90 on page 237

Hex	Opcode	Length	6502	65C02	65816	Cycles
28	PLP	1	⋆	⋆	⋆	4

All flags are replaced by the values in the byte pulled from the stack.

As this is an 8-bit register, only a single byte is pulled from the stack.

65816

On the 65816, the e bit which indicates 6502 emulation mode, is not included when the flags are pulled from the stack.

The only way to access the e flag is via the **XCE**[87] instruction.

87: **XCE** Section 16.90 on page 237

16.55 PLX Pull Index Register X

Format $X \leftarrow (STACK)$

Description Pull the X index register from the stack.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set if the most significant bit of the pulled value is set, else cleared

z Set if value pulled is zero, else cleared.

Hex	Opcode	Length	6502	65C02	65816	Cycles
FA	PLX	1		⋆	⋆	4

On all processors and the 65816 in 8-bit index register mode (x=1), a single byte is pulled from the stack.

On the 65816 in 16-bit index register mode (x=0), 2 bytes are pulled from the stack, low byte first then the high byte.

6502

Before **PLX** was introduced on the 65C02, this was implemented by pulling into A than transferring to XThe downside to this was it used two bytes and required the accumulator.

Listing 16.3: Implementing **PLX** on the 6502

```
; Pulling X from
; the stack
;
; 65C02 onwards
PLX
;
; 6502
PLA
TAX
```

16.56 PLY
Pull Index Register Y

Format $Y \leftarrow (STACK)$

Description Pull the Y index register from the stack.

Flags

	7	6	5	4	3	2	1	0	
	n	v		b	d	i	z	c	8-bit
	n	v	m	x	d	i	z	c	16-bit

n Set if the most significant bit of the pulled value is set, else cleared

z Set if value pulled is zero, else cleared.

Listing 16.4: Implementing **PLY** on the 6502

```
; Pulling Y from
; the stack
;
; 65C02 onwards
PLY
;
; 6502
PLA
TAY
```

Hex	Opcode	Length	6502	65C02	65816	Cycles
7A	PLY	1		⋆	⋆	4

On all processors and the 65816 in 8-bit index register mode (x=1), a single byte is pulled from the stack.

On the 65816 in 16-bit index register mode (x=0), 2 bytes are pulled from the stack, low byte first then the high byte.

6502

Before **PLY** was introduced on the 65C02, this was implemented by pulling into A than transferring to YThe downside to this was it used two bytes and required the accumulator.

16.57 REP
Reset Status Bits

Description REP

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

Hex	Opcode	Length	6502	65C02	65816	Cycles
C2	REP #	2			⋆	3

16.58 ROL
Rotate Memory or Accumulator Left

Function

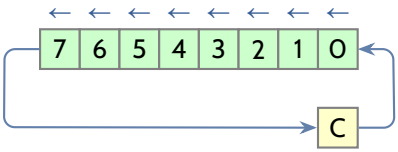


Figure 16.6: Rotate Left

Description Rotate the contents of the memory location left by one bit. Bit-7 is moved to the Carry Flag. The Carry Flag is moved to Bit-0.

On the 65816 with 16-bit accumulators (m=0), then the operation is 16-bit so Bit-15 moves to Carry.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n set if most significant bit of result is set

z result is zero

c The high bit becomes carry

Hex	Opcode	Length	6502	65C02	65816	Cycles
26	`ROL dp`	2	⋆	⋆	⋆	5
2A	`ROL A`	1	⋆	⋆	⋆	2
2E	`ROL addr`	3	⋆	⋆	⋆	6
36	`ROL dp,X`	2	⋆	⋆	⋆	6
3E	`ROL addr,X`	3	⋆	⋆	⋆	7

16.59 ROR Rotate Memory or Accumulator Right

Function

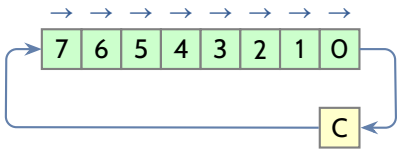


Figure 16.7: Rotate Right

Description Rotate the contents of the memory location right by one bit. Bit-0 is moved to the Carry Flag. The Carry Flag is moved to Bit-7.

On the 65816 with 16-bit accumulators (m=0), then the operation is 16-bit so the Carry moves to Bit-15.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n set if most significant bit of result is set

z result is zero

c The high bit becomes carry

Hex	Opcode	Length	6502	65C02	65816	Cycles
66	`ROR dp`	2	⋆	⋆	⋆	5
6A	`ROR A`	1	⋆	⋆	⋆	2
6E	`ROR addr`	3	⋆	⋆	⋆	6
76	`ROR dp,X`	2	⋆	⋆	⋆	6
7E	`ROR addr,X`	3	⋆	⋆	⋆	7

16.60 RTI
Return from Interrupt

Confirm the order of these on the stack as I have conflicting documentation on this

Description **RTI** pulls the status register and program counter from the stack causing the program to continue from where it was interrupted.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

All processors

c carry

d decimal mode

i IRQ interrupt disable

n negative result

v overflow

z zero result

8-bit & 65816 in emulation mode

b BRK caused interrupt

65816 in 16-bit native mode

m 8-bit memory/accumulator

x 8-bit index registers

65816 in either mode

e Emulation mode, not accessible other than via the **XCE**[88] instruction

88: **XCE** Section 16.90 on page 237

Hex	Opcode	Length	6502	65C02	65816	Cycles
40	RTI	1	⋆	⋆	⋆	6

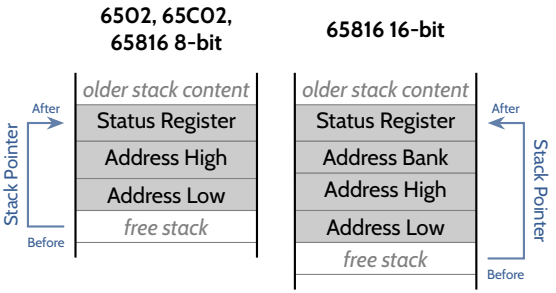


Figure 16.8: RTI stack usage

Stack format

All flags are replaced by the values in the byte pulled from the stack. The break flag is ignored.

Unlike **RTS**[89] or **RTL**[90], the address pulled from the stack is the exact address execution will be resumed from.

89: RTS Section 16.62 on page 205

90: RTL Section 16.61 on page 204

16.61 RTL
Return from long subroutine

Function

$$PC \leftarrow (STACK) + 1$$
$$PCB \leftarrow (STACK)$$

Description **RTL** returns from a subroutine that was called by the long version of **JSR**, that is with a 24-bit address.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

See also **JSR**[91], **RTS**[92]

91: **JSR** Section 16.32 on page 172
92: **RTS** Section 16.62 on page 205

Hex	Opcode	Length	6502	65C02	65816	Cycles
6B	RTL	1			⋆	6

It pulls the Program Counter from the stack, incrementing the value by one followed by the Program bank. The processor will then resume execution from the new address.

This address is usually the instruction after the original **JSR** long instruction which called the subroutine.

16.62 RTS
Return from subroutine

Function $PC \leftarrow (STACK) + 1$

Description **RTS** returns from a subroutine that was called by the non-long versions of **JSR**, that is with a 16-bit address.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

See also **JSR**[93], **RTL**[94]

93: **JSR** Section 16.32 on page 172

94: **RTL** Section 16.61 on page 204

Hex	Opcode	Length	6502	65C02	65816	Cycles
60	RTS	1	⋆	⋆	⋆	6

It pulls the Program Counter from the stack, incrementing the value by one. The processor will then resume execution from the new address.

This address is usually the instruction after the original **JSR** instruction which called the subroutine.

16.63 SBC
Subtract with Carry

Function $A \leftarrow (A) - DATA - \overline{C}$

Description Subtract the contents of memory or a literal value from the accumulator, subtracting one more if the carry flag is clear, and store the result in the accumulator.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n set if most significant bit of result is set

v set if signed overflow

z result is zero

c set if unsigned borrow not required, cleared if unsigned borrow.

See also **ADC**[95]

95: **ADC** Section 16.1 on page 134

Hex	Opcode	Length	6502	65C02	65816	Cycles
E1	SBC (dp,X)	2	⋆	⋆	⋆	6
E3	SBC sr,S	2			⋆	5
E5	SBC dp	2	⋆	⋆	⋆	3
E7	SBC [dp]	2			⋆	6
E9	SBC #	2	⋆	⋆	⋆	2
ED	SBC addr	3	⋆	⋆	⋆	4
EF	SBC long	4			⋆	5
F1	SBC (dp),Y	2	⋆	⋆	⋆	5
F2	SBC (dp)	2	⋆	⋆	⋆	5
F3	SBC (sr,S),Y	2			⋆	7
F5	SBC dp,X	2	⋆	⋆	⋆	4
F7	SBC [dp],Y	2			⋆	6
F9	SBC addr,Y	3	⋆	⋆	⋆	4
FD	SBC addr,X	3	⋆	⋆	⋆	4
FF	SBC long,X	4			⋆	5

Decimal mode

SBC is affected by the decimal flag so will either operate in binary or decimal mode dependent on that flag.

SUB Subtract without Carry

As there is no SUB instruction, if you want to Subtract without the carry, you need to use the **SEC** instruction prior to using **SBC**.

16.64 SEC
Set Carry

Function $C \leftarrow 1$

Description The carry flag is set.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

c Always set

Hex	Opcode	Length	6502	65C02	65816	Cycles
38	SEC	1	⋆	⋆	⋆	2

This is often required before an **SBC**[96] instruction to perform subtraction without carry.

96: **SBC** Section 16.63 on page 206

16.65 SED
Set Decimal Mode

Function $D \leftarrow 1$

Description The decimal flag is set setting decimal mode for the **ADC** and **SBC** instructions.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

d Always set

Hex	Opcode	Length	6502	65C02	65816	Cycles
F8	SED	1	⋆	⋆	⋆	2

16.66 SEI
Disable Hardware Interrupts

Function $I \leftarrow 1$

Description The interrupt flag is set disabling hardware interrupts.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

i Always set

Hex	Opcode	Length	6502	65C02	65816	Cycles
78	SEI	1	⋆	⋆	⋆	2

Some time-critical code, usually in realtime systems or games, wrap a section of code between **SEI** ...**CLI**[97] instructions so they get the processor to themselves.

97: **CLI** Section 16.18 on page 155

However, whilst the I flag is set, then it is possible that a hardware interrupt will go unnoticed if it's set for too long a period.

16.67 SEP Set Status Bits

Description SEP

Hex	Opcode	Length	6502	65C02	65816	Cycles
E2	SEP	2			⋆	3

16.68 STA
Store Accumulator to Memory

Function $M \leftarrow (A)$

Description Store the accumulator at the specified address.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

Hex	Opcode	Length	6502	65C02	65816	Cycles
81	STA (dp,X)	2	⋆	⋆	⋆	6
83	STA sr,S	2			⋆	4
85	STA dp	2	⋆	⋆	⋆	3
87	STA [dp]	2			⋆	6
8D	STA addr	3	⋆	⋆	⋆	4
8F	STA long	3			⋆	4
91	STA (dp),Y	2	⋆	⋆	⋆	5
92	STA (dp)	2		⋆	⋆	5
93	STA (sr,S),Y	2			⋆	7
95	STA dp,X	2	⋆	⋆	⋆	4
97	STA [dp],Y	2			⋆	6
99	STA addr,Y	3	⋆	⋆	⋆	4
9D	STA addr,X	3	⋆	⋆	⋆	4
9F	STA long,X	4			⋆	5

6502, 65C02 and 65816 in 8-bit accumulator mode (m=1)

The value stored takes only one byte of memory.

65816 in 16-bit accumulator mode (m=0)

The value stored takes two bytes of memory.

16.69 STP
Stop Processor

Function Stop the processor

Description **STP** will stop the processors oscillator input, shutting down the processor until a reset occurs by pulling the $\overline{RES}$ pin low.

Hex	Opcode	Length	6502	65C02	65816	Cycles
DB	STP	1			⋆	2

As power consumption is a function of frequency in CMOS circuits, stopping the clock cuts power to almost nothing.

16.70 STX
Store X to Memory

Function $M \leftarrow (X)$

Description Store the X index register to the specified memory address.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

Hex	Opcode	Length	6502	65C02	65816	Cycles
86	STX dp	2	⋆	⋆	⋆	3
8E	STX addr	3	⋆	⋆	⋆	4
96	STX dp,Y	2	⋆	⋆	⋆	4

6502, 65C02 and 65816 in 8-bit index register mode (x=1)

The value stored is 8-bits so takes only one byte of memory.

65816 in 16-bit index register mode (x=0)

The value stored is 16-bits so takes two bytes of memory.

16.71 STY
Store Y to Memory

Function $M \leftarrow (Y)$

Description Store the Y index register to the specified memory address.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

Hex	Opcode	Length	6502	65C02	65816	Cycles
84	STY dp	2	⋆	⋆	⋆	3
8C	STY addr	3	⋆	⋆	⋆	4
94	STY dp,X	2	⋆	⋆	⋆	4

6502, 65C02 and 65816 in 8-bit index register mode (x=1)

The value stored is 8-bits so takes only one byte of memory.

65816 in 16-bit index register mode (x=0)

The value stored is 16-bits so takes two bytes of memory.

16.72 STZ
Store Zero to Memory

Function $M \leftarrow 0$

Description Store zero at the specified address.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

Hex	Opcode	Length	6502	65C02	65816	Cycles
64	STZ dp	2		⋆	⋆	3
74	STZ dp,X	2		⋆	⋆	4
9C	STZ addr	3		⋆	⋆	4
9E	STZ addr,X	3		⋆	⋆	5

65C02 and 65816 in 8-bit index register mode (x=1)

The value stored takes only one byte of memory.

65816 in 16-bit index register mode (x=0)

The value stored takes two bytes of memory.

16.73 TAX
Transfer Accumulator to X

Function $X \leftarrow (A)$

Description Transfer the accumulator to the X index register.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set to the most significant bit of the value

z Set if value pulled is zero, else cleared.

See also **TXA**[98]

98: **TXA** Section 16.82 on page 229

Hex	Opcode	Length	6502	65C02	65816	Cycles
AA	TAX	1	⋆	⋆	⋆	2

16.73.1 Transfer size

99: **TAY** Section 16.74 on page 219
100: **TYA** Section 16.85 on page 232

This applies to both the **TAX**, **TAY**[99], **TXA** and **TYA**[100] instructions.

6502 and 65C02

On all 8-bit processors, the data transferred is 8-bits.

65816 in emulation mode or native 8-bit index (x=1)

On the 65816 in 6502 emulation mode, or native mode with 8-bit Index registers (x=1), the data transferred is 8-bit.

65816 in native mode with 16-bit index registers (x=0)

In 65816 native mode with 16-bit index registers (x=0), when the accumulator is also 16-bit (m=0) then A is transferred in full.

But if the accumulator is 8-bit (m=1) then the destination's lower 8-bits are from A whilst the upper 8-bits are from the hidden 8-bit B accumulator.

16.74 TAY
Transfer Accumulator to Y

Function $Y \leftarrow (A)$

Description Transfer the accumulator to the Y index register.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set to the most significant bit of the value

z Set if value pulled is zero, else cleared.

See also **TYA**[101]

101: **TYA** Section 16.85 on page 232

Hex	Opcode	Length	6502	65C02	65816	Cycles
A8	TAY	1	⋆	⋆	⋆	2

The size of the data transferred is described in section Subsection 16.73.1 on the facing page.

16.75 TCD
Transfer 16-Bit Accumulator to Direct Page Register

Function $DP \leftarrow (A_{16})$

Description This instruction transfers the 16-bit Accumulator to the Direct Page register.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set to the most significant bit of the value

z Set if value pulled is zero, else cleared.

See also **TDC**[102]

102: **TDC** Section 16.77 on page 222

Hex	Opcode	Length	6502	65C02	65816	Cycles
5B	TCD	1			⋆	2

If the accumulator is 8-bit (either m=1 or in 6502 emulation mode) then the lower 8-bits of the direct page register is set to A. The upper 8-bits come from the hidden 8-bit B accumulator.

Naming

The instruction is called **TCD** as it refers to the 16-bit A Accumulator as C because the operation is always 16-bit.

Some assemblers call this **TAD** so that it refers to the traditional Accumulator name.

16.76 TCS Transfer Accumulator to Stack Pointer

Function $SP \leftarrow (A_{16})$

Description This instruction transfers the 16-bit Accumulator to the stack pointer.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

Hex	Opcode	Length	6502	65C02	65816	Cycles
1B	TCS	1			⋆	2

The **TCS** and **TXS**[103] instructions are the only two that can change the stack pointer directly. They are also the only transfer instructions that do not alter the flags.

103: **TXS** Section 16.83 on page 230

In emulation mode only the lower 8-bits of the accumulator are transferred as the stack is confined to page one.

Otherwise, if the accumulator is 8-bit (either m=1) then the lower 8-bits of the stack pointer is set to A. The upper 8-bits come from the hidden 8-bit B accumulator.

With 16-bit accumulators, the entire accumulator is transferred.

Naming

The instruction is called **TCS** as it refers to the 16-bit A Accumulator as C because the operation is always 16-bit.

Some assemblers call this **TAS** so that it refers to the traditional Accumulator name.

16.77 TDC
Transfer Direct Page Register to 16-Bit Accumulator

Function $A_{16} \leftarrow (DP)$

Description This instruction transfers the Direct Page register to the 16-bit Accumulator.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set to the most significant bit of the value

z Set if value pulled is zero, else cleared.

See also **TCD**[104]

104: **TCD** Section 16.75 on page 220

Hex	Opcode	Length	6502	65C02	65816	Cycles
7B	TDC	1			⋆	2

If the accumulator is 8-bit (either m=1 or in 6502 emulation mode) then A is set to the lower 8-bits of the direct page register. The hidden 8-bit B accumulator receives the upper 8-bits.

Naming

The instruction is called **TDC** as it refers to the 16-bit A Accumulator as C because the operation is always 16-bit.

Some assemblers call this **TDA** so that it refers to the traditional Accumulator name.

16.78 TRB
Test and Reset Memory Bits against Accumulator

Function

$$M \leftarrow (M) \wedge \overline{(A)}$$
$$P_z \leftarrow (M) \wedge (A)$$

Description For each bit set in the accumulator, clear the associated bit at a given memory location. Once that is done, take the new value for the memory location, AND it with the accumulator and set the Z flag if the result is true. The accumulator is unchanged.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

z Set if the memory location after being modified, AND'ed with the accumulator is zero.

Hex	Opcode	Length	6502	65C02	65816	Cycles
14	TRB dp	2		⋆	⋆	5
1C	TRB addr	3		⋆	⋆	6

The size of memory affected is described in Subsection 16.78.1 on page 224.

Unlike **BIT**[105], **TRB** is a read-write operation where the memory is modified.

105: **BIT** Section 16.7 on page 141

16.78.1 Size of memory affected

For the 8-bit processors and the 65816 in either 6502 emulation mode or in 8-bit accumulator/memory mode (m=1), the values in the accumulator and memory are 8-bit.

For the 65816 in native 16-but accumulator/memory mode, the values in the accumulator and memory are 16-bit.

16.79 TSB
Test and Set Memory Bits against Accumulator

Function

$$M \leftarrow (M) \vee \overline{(A)}$$
$$P_z \leftarrow (M) \wedge (A)$$

Description For each bit set in the accumulator, set the associated bit at a given memory location. Once that is done, take the new value for the memory location, AND it with the accumulator and set the Z flag if the result is true. The accumulator is unchanged.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

z Set if the memory location after being modified, AND'ed with the accumulator is zero.

Hex	Opcode	Length	6502	65C02	65816	Cycles
04	TSB dp	2		⋆	⋆	5
0C	TSB addr	3		⋆	⋆	6

The size of memory affected is described in Subsection 16.78.1 on the following page

Unlike **BIT**[106], **TSB** is a read-write operation where the memory is modified.

106: **BIT** Section 16.7 on page 141

16.80 TSC
Transfer Stack Pointer to 16-Bit Accumulator

Function $A_{16} \leftarrow (SP)$

Description This instruction transfers the 16-bit stack pointer to the 16-bit Accumulator, regardless of the setting of the accumulator/memory mode flag m.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

Hex	Opcode	Length	6502	65C02	65816	Cycles
3B	TSC	1			⋆	2

65812 in 6502 emulation mode

In emulation mode the lower 8-bits of the stack pointer goes into the accumulator and 1 to the B accumulator, as the stack is confined to page one.

65812 in native mode

Otherwise, if the accumulator is 8-bit (either m=1) then the lower 8-bits of the stack page is put into the A accumulator. The upper 8-bits come go to the hidden 8-bit B accumulator.

With 16-bit accumulators (m=0), the stack pointer goes into the A accumulator.

Naming

The instruction is called **TSC** as it refers to the 16-bit A Accumulator as C because the operation is always 16-bit.

Some assemblers call this **TSA** so that it refers to the traditional Accumulator name.

16.81 TSX
Transfer Stack Pointer to X

Function $X \leftarrow (SP)$

Description This instruction transfers the stack pointer to the X index register. The stack pointer is not changed by the operation.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set to the most significant bit of the value

z Set if value pulled is zero, else cleared.

See also **TXS**[107]

107: **TXS** Section 16.83 on page 230

Hex	Opcode	Length	6502	65C02	65816	Cycles
BA	TSX	1	⋆	⋆	⋆	2

6502 and 65C02

On all 8-bit processors the 8-bits of the stack pointer is transferred.

65812 in 6502 emulation mode

On the 65816 in 6502 emulation mode, only the lower 8-bits of the stack pointer is transferred.

65812 in native mode

For the 65816 in native 8-bit index register mode, only the lower 8-bits of the stack pointer is transferred. The upper 8-bits are not transferred.

In 16-bit index register mode, the full 16-bits are transferred to the X register.

16.82 TXA
Transfer X to Accumulator

Format $A \leftarrow (X)$

Description Transfer the X index register to the accumulator

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set to the most significant bit of the value

z Set if value pulled is zero, else cleared.

See also **TAX**[108]

108: **TAX** Section 16.73 on page 217

Hex	Opcode	Length	6502	65C02	65816	Cycles
8A	TXA	1	⋆	⋆	⋆	2

The size of the data transferred is described in section Subsection 16.73.1 on page 218.

16.83 TXS
Transfer X to Stack Pointer

Function $SP \leftarrow (X)$

Description This instruction transfers the X index register to the stack pointer. The X register is not changed by the operation.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

See also **TSX**[109]

109: **TSX** Section 16.81 on page 228

Hex	Opcode	Length	6502	65C02	65816	Cycles
9A	TXS	1	⋆	⋆	⋆	2

The **TXS** and **TCS**[110] instructions are the only two that can change the stack pointer directly. They are also the only transfer instructions that do not alter the flags.

110: **TCS** Section 16.76 on page 221

In emulation mode only the lower 8-bits of the accumulator are transferred as the stack is confined to page one.

16.84 TXY
Transfer X to Y

Function $Y \leftarrow (X)$

Description Transfer the X index register to the Y index register. The value of X is unchanged.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set to the most significant bit of the value

z Set if value pulled is zero, else cleared.

See also **TYX**[111]

111: **TYX** Section 16.86 on page 233

Hex	Opcode	Length	6502	65C02	65816	Cycles
9B	TXY	1			⋆	2

As the sizes of the two registers will always be the same, the size of the data transferred will be that of the relevant mode, 8-bit when x=1 and 16-bit when x=0.

16.85 TYA
Transfer Y to Accumulator

Format $A \leftarrow (Y)$

Description Transfer the Y index register to the accumulator.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set to the most significant bit of the value

z Set if value pulled is zero, else cleared.

See also **TAY**[112]

112: **TAY** Section 16.74 on page 219

Hex	Opcode	Length	6502	65C02	65816	Cycles
98	TYA	1	⋆	⋆	⋆	2

The size of the data transferred is described in section Subsection 16.73.1 on page 218.

16.86 TYX
Transfer Y to X

Function $X \leftarrow (Y)$

Description Transfer the Y index register to the X index register. The value of Y is unchanged.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set to the most significant bit of the value

z Set if value pulled is zero, else cleared.

See also **TXY**[113]

113: **TXY** Section 16.84 on page 231

As the sizes of the two registers will always be the same, the size of the data transferred will be that of the relevant mode, 8-bit when x=1 and 16-bit when x=0.

16.87 WAI
Wait until hardware Interrupt

Description Halt the processor and wait for a hardware interrupt

Hex	Opcode	Length	6502	65C02	65816	Cycles
CB	WAI	1			⋆	3

The **WAI** instruction pulls the RDY pin low, reducing power consumption. RDY remains low until an external hardware interrupt (NMI, IRC, ABORT or RESET) is received.

16.88 WDM Reserved

Function Reserved for future use

Hex	Opcode	Length	6502	65C02	65816	Cycles
42	WDM	2			⋆	n

The 65802 and 654816 processors use 255 out of the possible 256–8-bit opcodes.

The remaining opcode is this one, labeled **WDM** which happens to be the initials of William D. Mensch who designed the processors.

To allow additional instructions to be added later this instruction act's as a prefix allowing an additional 256 opcodes.

This is a similar technique to the Z80 and 8080 processors which have 2-byte extension opcodes.



Do not use this instruction.

It will break if/when a future processor is released with additional instructions.

In addition, the byte and cycle count is subject to change in future processors.

16.89 XBA
Exchange the B and A Accumulators

Function $A \leftrightarrow B$

Description This instruction swaps the A and B accumulators.

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

n Set to the most significant bit of the new 8-bit value in the A accumulator

z Set if the new 8-bit value in the A accumulator is zero, else cleared.

Note The flags affected are based on the value of the A accumulator once the swap has been performed.

Here A represents the low-order byte of the C accumulator whilst the hidden B accumulator is the high-order byte.

C is commonly used to refer to the full 16-bit accumulator regardless of which mode the processor is in.

Swapping bytes in 16-bit value

XBA can be used to invert the two bytes of a 16-bit value, or temporarily store an 8-bit value from the A accumulator in B as in 8-bit mode B is not normally accessible.

The instruction is not affected by eiher of the mode select flags or the emulation flag.

Naming

Some assemblers support **SWA** as an alternate mnemonic to **XBA**.

16.90 XCE Exchange Carry and Emulation Bits

Function Switch between native and 6502 emulation mode

$P_e \leftrightarrow P_c$

Description The **XCE** instruction is the only means provided by the 65816 to switch between native and 6502 emulation modes.

Processor Mode The e flag defines the processor mode and takes the value of the carry flag in this operation.

e=0 The processor is in native mode

e=1 The processor is in 6502 emulation mode

Flags

7	6	5	4	3	2	1	0	
n	v		b	d	i	z	c	8-bit
n	v	m	x	d	i	z	c	16-bit

Entering native[114] mode

m Set to 1 if switching to native mode.

x Set to 1 when switching to native mode

c becomes the original value of the e flag.

Entering 6502[115] mode

b b[116] is only available in emulation mode

c becomes the original value of the e flag.

114: e=0

115: e=1

116: b and x share the same bit in the status register

Hex	Opcode	Length	6502	65C02	65816	Cycles
FB	XCE	1			⋆	2

The e flag is not visible by any means other than via this instruction.

16.90.1 Switching to 6502 emulation mode

To enter emulation mode you need to first set carry then execute **XCE**, causing the following effects:

- The m and x flags are removed from the status register.
- The b flag appears in the status register so that **BRK** interrupts can be identified.
- The accumulator becomes 8-bit keeping the low-order byte. The high-order byte is preserved as the hidden B accumulator.
- The index registers become 8-bit keeping the low-order byte. The high-order byte is lost.
- The stack pointer becomes 8-bit as the stack is restricted to page one.

Listing 16.5: Switch 65816 to 6502 emulation mode

```
1    SEC     ; Switch 65816 into 6502 emulation
     mode
2    XCE
```

16.90.2 Switching to native mode

To enter native mode, you need to clear carry then execute **XCE**, causing the following effects:

- The b flag is removed from the status register. This is not needed as **BRK** has it's own vector.
- The m and x flags appear in the status register and are both set to 1 for 8-bit mode.

Listing 16.6: Switch 65816 to native mode

```
1    CLC     ; Switch 65816 to native mode
2    XCE
```

The Z80 Microprocessor

Page intentionally blank

17 Registers

The Z80 has 208 bits of memory available as a set of eighteen 8-bit and four 16-bit registers.

Sixteen of the 8-bit registers are in two sets of eight and include the accumulator, status flags and six general purpose registers.

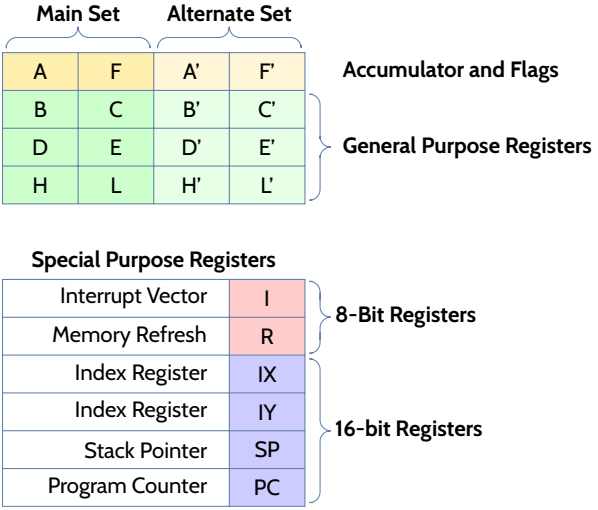


Figure 17.1: Z80 CPU Registers

The two of the 16-bit registers are used for pointing to the current instruction and the processor stack, whilst the other two are used in implementing lookup tables.

Of the final two 8-bit registers, one is for hardware use whilst the other for a specific type of interrupt. Both are rarely used by the programmer.

17.1 Accumulator and Flag registers

The CPU includes two independent 8-bit Accumulators and associated 8-bit Flag registers.

The Accumulator holds the results of 8-bit arithmetic or logical operations while the Flag Register indicates specific conditions for 8-bit or 16-bit operations, such as indicating whether the result of an operation is equal to 0.

The flags are discussed in length separately in Chapter 18 on page 249.

17.1.1 Using the Stack

You can only place both the Accumulator and Flag registers to the stack as a pair. It is not possible to push just the Accumulator or just the Flags.

To push to the stack, you need to use the **PUSH AF**[1] instruction.

To remove from the stack, you need to use the **POP AF**[2] instruction.

1: **PUSH AF** Section 22.44 on page 348

2: **POP AF** Section 22.43 on page 346

17.1.2 Selecting the alternate pair

You can swap the current Accumulator/Flags register pair with the alternate set with the **EX AF,AF'**[3] instruction.

Some uses for this is to quickly save state when handling either an interrupt or a call to a **RST** restart instruction.

3: **EX AF,AF'** Section 22.18 on page 302

17.2 General Purpose Registers

Two matched sets of general-purpose registers, each set containing six 8-bit registers, can be used individually as 8-bit registers or as 16-bit register pairs. One set is called BC, DE, and HL while the complementary set is called BC', DE', and HL'.

At any one time, the programmer can select either set of registers to work through a single exchange command (**EXX**[4]) for the entire set.

4: **EXX** Section 22.19 on page 304

In systems that require fast interrupt response, one set of general-purpose registers and an Accumulator/Flag Register can be reserved for handling this fast routine. When the routine begins it can then use the **EXX** instruction[5] to switch to the alternate set and at the end of the routine use **EXX** again.

5: You almost certainly want to use **EX AF,AF'** to swap AF with it's alternate at the same time

This process greatly reduces interrupt service time by eliminating the requirement for saving and retrieving register contents in the external stack during interrupt or subroutine processing as the **EXX** takes only 2 M cycles or 8 T states to switch them.

These general-purpose registers are used for a wide range of applications. They also simplify programing, specifically in ROM-based systems in which little external read/write memory is available.

17.2.1 Using the Stack

Each of the general purpose registers can be pushed to or pulled from the stack, but only as a register pair.

e.g. to push to the stack, you need to use the **PUSH**[6] instruction, whilst to remove from the stack, then you would use **POP**[7]. This applies to BC, DE or HL register pairs.

6: **PUSH** Section 22.44 on page 348

7: **POP** Section 22.43 on page 346

17.2.2 B, C and BC Registers

B Register

On its own, the B register is commonly used as an 8-bit counter.

The **DJNZ** [8] decrements B and performs a relative jump if it is not 0.

8: DJNZ means Decrement, Jump Non-Zero

I/O instructions like **IND**, **INI**, **OUTD** and **OUTI** decrement the B register so that it a loop can be terminated when B reaches 0.

Their repeat equivalents **INDR**, **INIR**, **OTDR** and **OTIR** will repeat themselves automatically until B reaches 0.

C Register

Many of the I/O instructions like **IN** and **OUT** as well as the block I/O instructions mentioned above use the C register to hold the I/O Port to read from or write to.

BC Register Pair

The Compare Memory and Load Memory Block instructions **CPD**, **CPI**, **LDD** and **LDI** are similar except they use the BC register pair as the counter, so they can handle up to 64K of memory instead of 256 bytes.

Similarly, their repeat equivalents **CPDR**, **CPIR**, **LDDR** and **LDIR** will repeat themselves automatically until BC reaches 0.

17.2.3 D, E and DE Registers

The Load Memory Block instructions **LDD**, **LDR**, **LDI** and **LDIR** use the DE register pair as the destination address to copy data into.

17.2.4 H, L and HL Registers

The HL register pair is almost exclusively used to refer to a memory address.

The Compare Memory and Load Memory Block instructions use the HL register pair as the address to read from.

However, most instructions that read from memory have a `(HL)` version[9] which either reads or writes to memory from the location pointed to by the HL pair.

9: In most instances, instructions support (HL) but do not support using the other general registers for addressing

17.3 Special Purpose Registers

There are six special purpose registers available, four 16-bit and two 8-bit registers. These registers do not have an alternate set available.

17.4 PC Register

The program counter holds the 16-bit address of the current instruction being fetched from memory. The Program Counter is automatically incremented after its contents are transferred to the address lines. When a program jump occurs, the new value is automatically placed in the Program Counter, overriding the incrementer.

17.5 SP Register



Unlike the 6502 which is, in the 8-bit equivalents to the Z80 limited to using page `0x01` and 256 bytes for it's stack.

The Z80 does not have this restriction.

The stack pointer holds the 16-bit address of the current top of a stack located anywhere

in external system RAM memory. The external stack memory is organized as a last-in first-out (LIFO) file.

Data can be pushed onto the stack from specific CPU registers or popped off of the stack to specific CPU registers through the execution of **PUSH**[10] and **POP**[11] instructions. The data popped from the stack is always the most recent data pushed onto it. The stack allows simple implementation of multiple level interrupts, unlimited subroutine nesting and simplification of many types of data manipulation.

10: **PUSH** Section 22.44 on page 348

11: **POP** Section 22.43 on page 346

17.6 IX and IY Registers

The two independent index registers hold a 16-bit base address that is used in indexed addressing modes. In this mode, an index register is used as a base to point to a region in memory from which data is to be stored or retrieved. An additional byte is included in indexed instructions to specify a displacement from this base. This displacement is specified as a two's complement signed integer. This mode of addressing greatly simplifies many types of programs, especially when tables of data are used.

17.6.1 Using the Stack

Like the 16-bit general purpose registers, the index register can be pushed to or pulled from the stack with the **PUSH** and **POP** instructions. This applies to the IX and IY registers.

17.7 I Interrupt Page Address Register

The Z80 CPU can be operated in a mode in which an indirect call to any memory location can be achieved in response to an interrupt.

The I register is used for this purpose and stores the high-order eight bits of the indirect address while the interrupting device provides the lower eight bits of the address. This feature allows interrupt routines to be dynamically located anywhere in memory with minimal access time to the routine.

The **IM 2**[12] instruction is used to select interrupt mode 2 which uses this register.

12: IM 2 Section 22.21 on page 306

17.8 R Memory Refresh Register

The Z80 CPU contains a memory refresh counter, enabling dynamic memories to be used with the same ease as static memories.

Seven bits of this 8-bit register are automatically incremented after each instruction fetch. The eighth bit remains as programmed, resulting from an **LD R, A**[13] instruction.

13: LD R, A Section 22.30 on page 320

The data in the refresh counter is sent out on the lower portion of the address bus along with a refresh control signal while the CPU is decoding and executing the fetched instruction.

This mode of refresh is transparent to the programmer and does not slow the CPU operation. The programmer can load the R register for testing purposes, but this register is normally not used by the programmer.

During refresh, the contents of the I Register are placed on the upper eight bits of the address bus.

Page intentionally blank

18 Flags

The Flag registers, F and F', supply information to the user about the status of the Z80 CPU at any particular time. Table 18.1 on page 249 shows the available flags.

Each of these two Flag registers contains 6 bits of status information that are set or cleared by CPU operations; bits 3 and 5 are not used.

Four of these bits (C, P/V, Z, and S) can be tested for use with conditional **JP**, **JR**, **CALL**, or **RET** instructions.

The H and N flags cannot be tested and are used for BCD arithmetic.

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Flag	S	Z	–	H	–	P/V	N	C

Table 18.1: Z80 Flag Register

18.1 C Carry Flag

The Carry Flag (C) is set or cleared depending on the operation being performed.

For addition instructions that generate a carry, and for subtraction instructions that generate a borrow, the Carry Flag is set.

The Carry Flag is reset by an addition instruction that does not generate a Carry, and by a subtraction instruction that does not generate a borrow. This saved Carry facilitates software routines for extended precision arithmetic.

Additionally, the **DAA** instruction sets the Carry Flag if the conditions for making the decimal adjustment are met.

For the logical instructions **AND**, **OR**, and **XOR**, the Carry flag is reset.

18.1.1 Rotations and the Carry Flag

For the **RLA**, **RRA**, **RLS**, and **RRS** instructions, the Carry bit is used as a link between the least-significant byte (LSB) and the most-significant byte (MSB) for any register or memory location.

18.1.2 Shifts and the Carry Flag

During left shift operations like **RLCA**, **RLC**, and **SLA**, the Carry flag contains the final value shifted out of bit 7 of any register or memory location.

Right shift operations like **RRCA**, **RRC**, **SRA**, and **SRL**, the Carry flag contains the final value shifted out of bit 0 of any register or memory location.

18.1.3 Setting the Carry Flag

The Carry flag can be set by the Set Carry Flag **SCF** instruction.

The **CCF** instruction will complement the Carry Flag, *e.g.* if it was set then reset it or set if it was originally reset.

18.1.4 Clearing the Carry Flag

Unlike the 6502, the Z80 has no dedicated Clear Carry Flag instruction.

However, the 6502's **CLC**[1] instruction can be emulated using operations like **AND A** or **OR A** which resets the carry flag but do not change the accumulator[2].

Using one of these instructions to clear the flag takes up less memory and take fewer cycles to execute than the other option which is to use **SCF** to set then **CCF** to compement the flag giving us a cleared carry flag.

1: **CLC** Section 16.16 on page 153

2: The Spectrum 48K ROM uses both **AND A** and **OR A** instructions to clear the carry flag.

18.2 Z Zero Flag

The Zero Flag (Z) is set (1) or cleared (0) if the result generated by the execution of certain instructions is 0.

Z	State
0	Result Not Zero
1	Result Zero

For 8-bit arithmetic and logical operations, the Z flag is set to a 1 if the resulting byte in the Accumulator is 0. If the byte is not 0, the Z flag is reset to 0.

For Compare (search) instructions, the Z flag is set to 1 if the value in the Accumulator is equal to the value in the memory location indicated by the value of the register pair HL.

When testing a bit in a register or memory location, the Z flag contains the complemented state of the indicated bit in the **BIT** instruction.

When reading or writing a byte between a memory location and ab I/O device using a **INI**, **IND**, **OUTI**, or **OUTD** instruction, if the result of decrementing Register B is 0, then the Z flag is set; otherwise it is cleared. Additionally, for byte inputs from I/O devices using **IN r,(C)**, the Z flag is set to indicate a 0-byte input.

18.3 S Sign Flag

S	State
0	Result Positive
1	Result Negative

The total range for two's-complement numbers is −128 ··· + 127

The Sign Flag stores the state of the most-significant bit of the Accumulator (bit 7).

When the Z80 CPU performs arithmetic operations on signed numbers, the binary twos-complement notation is used to represent and process numeric information. A positive number is identified by a 0 in Bit 7. A negative number is identified by a 1.

When inputting a byte from an I/O device to a register using an **IN r,(C)** instruction, the S Flag indicates either positive (S = 0) or negative (S = 1) data.

18.4 P/V Parity/Overflow flag

The Parity/Overflow (P/V) Flag is set to a specific state depending on the operation being performed.

18.4.1 Parity

P	Parity
0	Parity Odd
1	Parity Even

This flag is used with logical operations and rotate instructions to indicate the resulting parity is even.

The number of set bits in a byte are counted.

- If the total is odd, then odd parity is flagged *i. e.* P = 0
- If the total is even, then even parity is flagged *i. e.* P = 1

18.4.2 Arithmetic Overflow

For arithmetic operations, this flag indicates an overflow condition when the result in the Accumulator is greater than the maximum possible number (+127) or is less than the minimum possible number (−128).

This overflow condition is determined by examining the sign bits of the operands. For addition, operands with different signs never cause overflow. When adding operands with similar signs and the result contains a different sign, the Overflow Flag is set, as shown in the following example. The two numbers added together

+120 = 0111	1000	ADDEND	
+105 = 0110	1001	AUGEND	
+225 = 1110	0001	-95	$\overline{SUM}$

result in a number that exceeds +127 and the two positive operands result in a negative number −95, which is incorrect. The Overflow Flag is therefore set.

For subtraction, overflow can occur for operands of unalike signs. Operands of alike signs never cause overflow, as shown in the following example. The minuend sign

	+127	0111	1111	MINUEND
-	−64	1100	0000	SUBTRAHEND
	+191	1011	1111	DIFFERENCE

has changed from a positive to a negative, resulting in an incorrect difference; the Overflow Flag is set.

Another method for identifying an overflow is to observe the Carry to and out of the sign bit. If there is a Carry in and no Carry out, or if there is no Carry in and a Carry out, then an Overflow has occurred.

18.5 Lesser used flags

These flags are not used by user code. In fact there is no direct means of testing them, hence they are listed here, so you know what they are for, and can then ignore them.

18.5.1 H Half Carry Flag

The Half Carry Flag (H) is set or cleared depending on the Carry and Borrow status between bits 3 and 4 of an 8-bit arithmetic operation. This flag is used by the Decimal Adjust Accumulator **DAA** instruction to correct the result of a packed BCD add or subtract operation.

Table 18.2: Half Carry Flag states

H	Add	Subtract
1	Carry bit 3 to 4	Borrow from bit 4
0	No carry or borrow	

18.5.2 N Add/Subtract Flag

The Add/Subtract Flag (N) is used by the Decimal Adjust Accumulator instruction **DAA** to distinguish between the addition and subtraction instructions. It is cleared by all addition instructions and set by all subtraction instructions.

19 Arithmetic and Logic

The Z80 provides instructions for handling arithmetic, logic and the shifting of bits. All of these instructions act on the accumulator and either a register, or a memory location as the operand. The register could be any 8-bit register, the memory location pointed to the HL register pair, or an offset from one of the index registers IX or IY.

Some instructions can take a constant 8-bit immediate value, whilst some support 16-bit arithmetic which have to use one of the three general register pairs, IX, IY or the Stack Pointer.

19.1 Increment and Decrement

The **INC**[1] and **DEC**[2] instructions will increment or decrement a register, register pair or memory location by one.

Only the 8-bit operations affect the flags with Z set if the result is zero and s set if bit-7 is set. 16-bit operations against a register pair does not affect the flags.

1: **INC** Section 22.23 on page 310

2: **DEC** Section 22.14 on page 296

19.2 Negation and Complement

The **NEG**[3] instruction negates the value of the accumulator but in a two's complement friendly manner. This is the same as $A = 0 - A$. In other words negating 0 returns zero, but negating `0x01` returns `0xFF` which is -1 in two's complement.

The **CPL**[4] instruction complements the accumulator, that is it inverts every bit in the accumulator.

Figure 19.1 shows the result of both **NEG** and **CPL** on the value `0x01`

Figure 19.1: NEG and CPL

	Accumulator			
before	`0000 0001`	`0x01`	1
NEG	`1111 1111`	`0xFF`	255
CPL	`1111 1110`	`0xFE`	254

3: **NEG** Section 22.35 on page 335

4: **CPL** Section 22.12 on page 293

19.3 Addition and Subtraction

There are four instructions for addition and subtraction, two which use the Carry flag and two which do not.

Unlike the 6502, the Z80 provides instructions for handling both 8-bit and 16-bit addition and subtraction, so you can perform 16-bit arithmetic in a single instruction rather than having to handle each byte and the carry flag between them.

However, if you want to do 32-bit arithmetic, then you still have to deal with the carry flag, but less frequently as it's at the 16-bit level rather than every 8-bits.

19.3.1 With Carry or Borrow

The two instructions that use the carry flag are **ADC**[5] and **SBC**[6].

These instructions will add or subtract the operand along with the carry flag from the accumulator and place the result into the accumulator.

These are usually used with multiple-byte arithmetic where the carry is being provided by an earlier instruction.

5: **ADC** Section 22.1 on page 276
6: **SBC** Section 22.60 on page 375

19.3.2 Without Carry or Borrow

The two instructions that do not use the carry flag are **ADD**[7] and **SUB**[8].

These instructions simply add or subtract the operand from the accumulator and place the result into the accumulator. The Carry flag is still used, as it holds the carry or borrow of the result.

7: **ADD** Section 22.2 on page 278
8: **SUB** Section 22.66 on page 388

19.4 Compare

The **CP**[9] instruction will compare the source register, memory location or an immediate value against the accumulator. This is accomplished by subtracting the operand from the accumulator and discarding the result. Only the flags are effected which enable the result to be acted upon.

9: **CP** Section 22.7 on page 286

19.4.1 Branching on results

There are six possible comparisons that can be performed based on the differences between the accumulator and the operand. These can then be used with the **JR**[10], **JP**[11], **CALL**[12] and **RET**[13] instructions.

10: **JR** Section 22.29 on page 319

11: **JP** Section 22.28 on page 317

12: **CALL** Section 22.5 on page 283

13: **RET** Section 22.46 on page 354

Equals = and Not Equals ≠

These are the two obvious conditions and the most common. These both use the z flag, which is set if $A - s = 0$.

So if the accumulator equals the operand then the z for the relevant instructions would be `z`. Otherwise, `nz`.

```
1         LD A, 42
2         LD B, 32
3         CP B              ; Compare A and B
4         JR Z, END         ; Jump if A=B
5         JR NZ, NOTEQ      ; Jump if A != B
6
7   ; Destination for not equal
8   .NOTEQ  RET
9
10  ; Destination when equal
11  .END    RET
```

Listing 19.1: Example of equals and not-equals

Line 3: Effectively A-B

Line 4: Jump is $A \neq B$

Line 5: Jump is $A = B$

Less Than <

For the accumulator to be less than the operand, the result of $A - s$ will be negative. In the flags, this is represented by the s flag which will be set when the result is negative. Specifically, if bit-7 of the result is set then the s flag is set.

For 8-bit two's complement arithmetic this is valid as long as the difference is less than 128. If the result is outside that then the result will be positive.

```
1         LD A, 32
2         LD B, 42
3         CP B          ; Compare A and B
4         JR M, END     ; Jump if A <  B
5
6 ; Code run when A >= B
7
8 ; Destination when less than
9 .END    RET
```

Listing 19.2: Example of less than
Line 3: Effectively A-B

Line 4: Jump if $A < B$

Greater Than or Equals ≥

You might be wondering why this one is next? This is because it is infact the opposite of Less than, so this is the case if the s flag is clear the result is Positive. Specifically, if bit-7 of the result is clear then the s flag is also clear.

As zero is also positive in two's complement, then the ≥ operation is simply a check against the s flag.

```
1         LD A, 32
2         LD B, 42
3         CP B          ; Compare A and B
4         JR P, END     ; Jump if A >= B
5
6 ; Code run when A < B
7
8 ; Destination when greater than or equal
9 .END    RET
```

Listing 19.3: Example of less than and greater than or equals
Line 3: Effectively A-B

Line 4: Jump if $A \geq B$

Less Than or Equals ≤

Less Than or Equals is a bit more complicated as it can only be represented by a combination of the Z and S flags. If we are less than, then we can look for $s = 1$, but we also need to look for equality when $z = 1$.

```
1         LD A, 32
2         LD B, 42
3         CP B          ; Compare A and B
4         JR M, END     ; Jump if A < B
5         JR Z, END     ; Jump if A = B
6
7  ; Code run when A > B
8
9  ; Destination when less than
10 .END    RET
```

Listing 19.4: Example of less than or equals

Line 3: Effectively A-B

Line 4: This would jump whenever $A < B$

Line 5: If that didn't trigger then jump $A = B$

Greater Than >

This has the same problem as Less Than or Equals. Here we need to check if it is positive but also that it is not equal to zero.

```
1         LD A, 42
2         LD B, 32
3         CP B          ; Compare A and B
4         JR Z, NOTGT   ; Jump if A = B
5         JR P, END     ; Jump if A >= B
6  .NOTGT               ; Point we jump if A=B
7
8  ; Code run when A <= B
9
10 ; Destination when greater than
11 .END    RET
```

Listing 19.5: Example of greater than

Line 3: Effectively A-B

Line 4: Skip next check if $A = B$

Line 5: As $A \neq B$ due to the previous line then jump if $A > B$

19.5 Logic

The **AND**, **OR** and **XOR** instructions provide logic operations against the accumulator.

14: **AND** Section 22.3 on page 280

AND[14] performs a bit AND of the operand and the accumulator. The result is that the bits set in both the accumulator and the operand will be set in the result. All others will be clear.

	128	64	32	16	8	4	2	1	
	1	1	0	1	1	0	1	0	0xDA
AND	0	1	0	0	0	1	1	0	0x45
Result	0	1	0	0	0	0	1	0	0x42

15: **OR** Section 22.37 on page 337

OR[15] performs a bit OR of the operand and the accumulator. Here if a bit is set in either the accumulator or the operand then it will be set in the result. A clear bit in the result occurs if it is clear in both the accumulator and the operand.

	128	64	32	16	8	4	2	1	
	1	1	0	1	1	0	1	0	0xDA
OR	0	1	0	0	0	1	1	0	0x45
Result	1	1	0	1	1	1	1	0	0xDE

16: **XOR** Section 22.67 on page 390

XOR[16] performs an exclusive-or operation. Here, a bit is set in the result if it is set in either the accumulator or the operand - but not if it is set in both. If the bit is clear in both the accumulator and the operand then it is clear in the result.

	128	64	32	16	8	4	2	1	
	1	1	0	1	1	0	1	0	0xDA
XOR	0	1	0	0	0	1	1	0	0x45
Result	1	0	0	1	1	1	0	0	0x9C

19.6 Rotates and Shifts

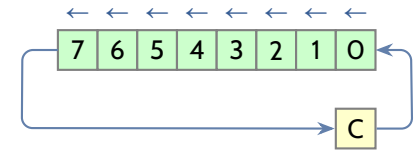
There are a group of instructions which shift the contents of a register one bit to the left or one bit to the right.

For each direction, there's the option of either inserting the carry flag into the register, a zero, or the branch-carry rotates which use the bit shifted out back into the register as well as c.

19.6.1 Shift and Rotate Left

These operations shift one bit to the left. This effectively multiplies the register by 2.

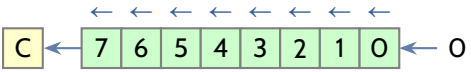
RL[17] and **RLA**[18] rotate bit-7 into the carry flag. The original carry flag is placed into bit-0.



17: **RL** Section 22.49 on page 357

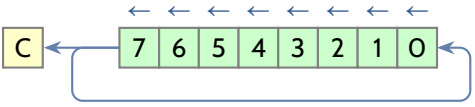
18: **RLA** Section 22.50 on page 359

SLA[19] is similar to **RL** except that 0 is placed into bit-0.



19: **SLA** Section 22.63 on page 382

RLC[20] and **RLCA**[21] rotates bit-7 into bit-0 as well as the carry flag.



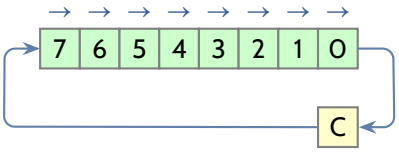
20: **RLC** Section 22.51 on page 360

21: **RLCA** Section 22.52 on page 362

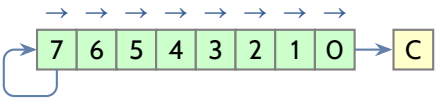
19.6.2 Shift and Rotate Right

These operations shift one bit to the right. This effectively divides the register by 2.

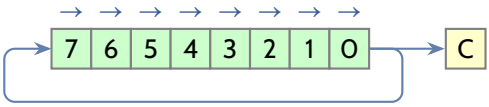
RR[22] and **RRA**[23] rotate bit-0 into the carry flag. The original carry flag is placed into bit-7.



SRA[24] is similar to **RR** except that, whilst bit-7 is shifted into bit-6, bit-7 itself is unchanged. This means that if you shift a negative number (in two's complement) it remains negative.



RRC[25] and **RRCA**[26] rotate bit-0 into bit-7 as well as the carry flag.



22: **RR** Section 22.54 on page 364

23: **RRA** Section 22.55 on page 366

24: **SRA** Section 22.64 on page 384

25: **RRC** Section 22.57 on page 368

26: **RRCA** Section 22.56 on page 367

19.7 Bit Manipulation

The Z80 provides 240 instructions for the manipulation of individual bits, split into three instructions.

The **BIT**[27] instruction tests the current state of a bit in a register or memory location, whilst the **SET**[28] and **RES**[29] instructions either set or reset an individual bit respectively.

All three accept the bit number affected and the location of that bit. The location can be the Accumulator, any of the individual general purpose registers, the memory location pointed to by the HL register pair, or the memory location at an offset from either of the IX or IY registers.

Only the **BIT** instruction affects the flags. The Z flag takes on the value of the bit being tested.

27: **BIT** Section 22.4 on page 281

28: **SET** Section 22.62 on page 378

29: **RES** Section 22.45 on page 350

19.8 Binary Coded Decimal

I only mention BCD here as, in the 45+ years that I've been writing assembly language I've never needed to deal with BCD arithmetic.

19.8.1 BCD Adjustment

30: DAA Section 22.13 on page 294

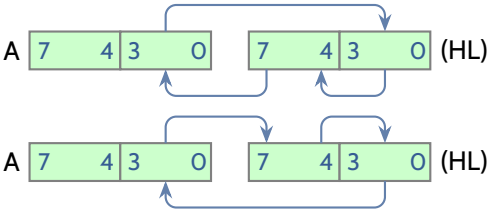
The **DAA**[30] instruction adjusts the accumulator after an arithmetic instruction to ensure the result is a valid BCD (Binary Coded Decimal) value.

19.8.2 BCD Rotate Left and Right

31: RLD Section 22.53 on page 363

32: RRD Section 22.58 on page 370

The **RLD**[31] and **RRD**[32] instructions rotate left and right both the accumulator and operand by the nibble rather than by bit. This enables for BCD digits to be rotated.



20 Input/Output

The majority of 8-bit microprocessors use memory-mapped I/O, where devices are attached directly to the address bus just like either ROM or RAM memory devices. Although this makes designs simpler it has the downside of using up a chunk of the available address space.

The Z80 can use this technique, but it also provides a separate bus where devices can exist without encroaching on the main memory space. This IO bus shares the same pins as the main Address Bus, but a separate pin is used to indicate that the address bus holds a device address rather than a memory address.

The advantages of this system is that the devices are separate from the main address space. The downside, however, is that only dedicated instructions can access this space.

This isn't the worst thing about this bus. Although the bus is 16-bits with an addressable 64K, in effect there are actually only 256 ports available for devices. When some peripherals can use up to 16 address[1] each, you could easily run out of ports.

1: The 6522VIA chip for the 6502 uses 16 bytes of address space

This issue will become evident later in Section 20.3 on page 268 where we discuss the problem with the upper portion of the address bus, then in Listing 20.1 on page 267 we show a real-life example where the use of the upper address bits of the port were "abused" to implement a keyboard scanner.

20.1 Input from a device

2: **IN** Section 22.22 on page 308

For reading from a device, the Z80 provides the **IN**[2] instruction to read a single byte from a device port. This comes in two versions.

IN A,(n)

This version takes an 8-bit port number as immediate data and placed it in the lower 8-bits of the address bus, providing access to up to 256 ports. The current value of the accumulator is placed in the upper 8-bits of the address bus. The byte is then read from that port using the data bus and placed into the accumulator.

IN r,(C)

This version takes the 8-bit port number from the C register and reads a byte from that port and places it into the 8-bit register r. Although the instruction mentions only `(C)`, the B register is also placed on the upper 8-bits of the address bus.

20.1.1 Reading from Port Example

Listing 20.1 on the next page shows a real-life example of reading from a port. It is the routine from the 48K ZX Spectrum ROM which scans the keyboard for any pressed keys. The keyboard is read by reading from the ULA custom chip located on port `0xFE`. When reading from the ULA, the upper 8-bits of the address bus are used to select the row on the keyboard.

On line 3, The **LD BC,0xFEFE** sets the BC register pair. C is the ULA port `0xFE` whilst B is set as a "counter" as well as the initial upper-half of the address bus when we read from the port.

The **IN A,(C)** on line 4 reads from the port and puts the value into the accumulator.

We then complement it with the **CPL** on line 5 as the hardware sets the bit to 0 when a key is pressed and we want a 1. The **AND 0x1F** on line 6 then masks out the top 3 bits which are nothing to do with the keyboard, and gives us 5 bits each representing a key on the current row.

The key is then decoded, which is out of scope for this example.

Finally, the **RLC B** on line 14 rotates the B register left. More specifically, this rotates the 0 bit in B left one place selecting the next keyboard row. As only one bit is clear in B, when that bit rotates into carry it causes the **JR C** on line 15 not to restart the scan.

Listing 20.1: Scan keyboard

```
1  KEYSCAN LD  L,0x2F      ; Initial key line, of 8
2          LD  DE,0xFFFF   ; Set to no key
3          LD  BC,0xFEFE   ; C=port 0xFE, B=counter
4  KEYLINE IN  A,(C)       ; Read from port
5          CPL             ; Complement so 1 is pressed, 0 not
6          AND 0x1F        ; Bits 0--4 are the keys in the row
7          JR  Z,KEYDONE   ; Jump if none of 5 keys in line pressed
8          LD  H,A         ; keybits to H
9          LD  A,L         ; get initial key
10 KEYBITS SUB 0x08        ; repeat sub 8 until a key bit found
11         SRL H
12         JR  NC,KEYBITS
13 KEYDONE DEC L           ; Line scanned so reduce for next pass
14         RLC B           ; Counter shifted
15         JR  C, KEYLINE  ; Jump while lines still to be scanned
16                         ; Test found keys
17         LD  A,D         ; Accept pair if D key is caps shift
18         RET Z
19         CP  0x19        ; Accept pair if D key is symbol shift
20         RET Z
21         LD  A,E         ; Check possibility E key is a
22         LD  E,D         ; symbol shift
23         LD  D,A
24         CP  0x18
25         RET
```

20.2 Output to a device

3: **OUT** Section 22.40 on page 341

To write to a device, there is the **OUT**[3] instruction which, like **IN**, writes a single byte to the device. It also comes with two versions.

OUT (n),A

This is the same as **IN A,(n)** in that it takes an 8-bit port number as immediate data and placed it in the lower 8-bits of the address bus, providing access to up to 256 ports. The current value of the accumulator is placed in the upper 8-bits of the address bus. The byte is then read from that port using the data bus and placed into the accumulator.

OUT (C),r

This is the same as **IN r,(C)** where it takes the 8-bit port number from the C register and reads a byte from that port and places it into the 8-bit register r. Although the instruction mentions only `(C)`, the B register is also placed on the upper 8-bits of the address bus.

20.3 The number of ports problem

As you have seen, the **IN** and **OUT** instructions formally only use the lower 8-bits of the address bus, giving us just 256 ports.

In Subsection 20.1.1 on page 266 show the use of the B register to select which keyboard row to scan even though device `0xFE` which is the official port selected.

So if those instructions also include either the Accumulator or the B register to the upper 8-bits then, as long as we remember to set those registers first why are we still limited to an effective 256 usable ports?

Well, the problem is that you cannot guarantee what is written to the upper byte of the address bus.

While we have the generic **IN** and **OUT** instructions, there are eight more available. These are the Block IO instructions which enable a block of memory to be either written to or read from a port.

These instructions use the C register to define the port to use, and the HL register pair the memory location to read or write from. However, they also use the B register to hold how many bytes to transfer.

So, during this block transfer, the B register, even though it's being written to the upper 8-bits of the address bus during the transfer, it is actually being decremented for each byte being transferred meaning it changes for each byte.

Section 21.3 on page 273 covers this process for reading from the device, but this isn't the worst of the problem. When reading from the device, B is decremented *after* the read, so if you are reading 5 bytes then, while performing the read the upper half of the address bus shows 5, then 4, 3, 2 and finally 1.

However, when writing to a device, B is decremented *before* the write, not after like during a read. This means that, when writing to the device, the upper half of the address bus sees 4, then 3, 2, 1 and finally 0.

Here is the problem.

If we use generic IO then we can set the upper 8-bits of the address bus to a value to give ourselves 64K of ports[4].

But if we use the block transfer instructions then we are limited to just 256 just because the upper bits change on each read or write and are not identical for either operation.

That said, this has not stopped anyone using this feature in the past, and it was common that the port number in C selected a device, but the upper 8 bits were used to

4: Or 256 devices each with 256 bytes of address space

The issue with the block commands didn't appear as they just not used that often.

In 45+ years, I've not seen those block IO instructions used in actual code.

select the address within the device. This is why Subsection 20.1.1 on page 266 works, because the designers limited themselves to the standard **IN A,(C)** instruction.

21 Block instructions

There are four sets of instructions which operate over a block of memory.

Each set consists of four instructions, two which increment though memory so operate from the start of the block, and two which decrement through the memory so operate from the end of the block.

21.1 Block Load

The block load instructions will copy up to 64K bytes of memory from one location to another.

These instructions use the HL register to hold the source address to read from, the DH register the destination address to write to, and the BC register to hold how many bytes to copy.

The **LDI**[1] instruction will copy the byte from the memory at HL and store it at DE. It then increments both DE and HL so the next time the instruction is executed it writes to the next address. The **LDIR**[2] instruction is identical, except that it will repeat the operation BC times.

The **LDD**[3] instruction is similar to **LDI** except that it decrements both DE and HL, so it moves down in memory. **LDDR**[4], like **LDIR** will repeat **LDD** BC times.

Of these instructions, the **LDIR** instruction is the most used one, commonly used in games to move graphics around in memory.

If BC is set to 0 before one of these instructions then the entire 64K of address space will be copied.

1: **LDI** Section 22.33 on page 333

2: **LDIR** Section 22.34 on page 334

3: **LDD** Section 22.31 on page 330

4: **LDDR** Section 22.32 on page 332

21.2 Compare Memory

The compare memory instructions searches a block of memory until it finds the first address which matches the value of the accumulator.

These instructions use the HL register to hold the address to check, and the BC register as a counter for the size of the block of memory.

If BC is set to 0 before one of these instructions then the entire 64K of address space will be searched.

21.2.1 Compare from start of block

To search from the start of a memory block, first set HL to the start address, BC with the number of bytes to check and A with the value to search for. The **CPI**[5] instruction can then be used to test the address HL points to with the accumulator.

5: **CPI** Section 22.10 on page 291

The **CPIR**[6] instruction is identical to **CPI** except that if there was no match it then repeats the instruction until either it finds a result or it reaches the end of the block.

6: **CPIR** Section 22.11 on page 292

21.2.2 Compare from end of block

To search from the end of a block of memory then you would set HL to the end address of the memory block. Like comparing from the start, BC contains the size of the block to search and A the value to search for.

You would then use the **CPD**[7] instruction to do the comparison. **CPD** is the same as **CPI** except that it decrements HL each time it runs.

7: **CPD** Section 22.8 on page 288

The **CPDR**[8] instruction is used to repeat the operation automatically until it finds a match or reaches the start of the memory block.

8: **CPDR** Section 22.9 on page 290

21.3 Block Input

The block input instructions will read up to 256 bytes of memory from a peripheral device in one go.

If B is set to 0 before one of these instructions then 256 bytes will be read from the device.

These instructions use the HL register to hold the address to write to, the C register to hold the device's port and the B register to hold how many bytes to read.

The **INI**[9] instruction will read from the device and write to memory at HL. It then increments HL so the next time the instruction is executed it writes to the next address. The **INIR**[10] instruction is identical, except that it will repeat B times automatically.

The **IND**[11] instruction is similar to **INI** except that it decrements HL, so it moves down in memory. **INDR**[12], like **INIR** will repeat **IND** B times.

9: **INI** Section 22.26 on page 315

10: **INIR** Section 22.27 on page 316

11: **IND** Section 22.24 on page 312

12: **INDR** Section 22.25 on page 314

21.3.1 Address Bus

Note that, when accessing the port, the value of C is placed on bits $A_0 \ldots A_7$ of the address bus whilst the value of the B register is placed on bits $A_8 \ldots A_{15}$.

Although this would infer that you could have upto 65536 addressable ports, remember that the B register holds the count of the number of bytes to read from the port.

This means you are still restricted to a useful 256 I/O ports to read from as the upper byte will change for each byte read.

You should also look at Subsection 21.4.1 on the following page as the output block instructions also differ in how they use the address bus.

21.4 Block Output

Writing a block of memory to a peripheral is identical to block input. The HL register pair holds the memory location to read from, the C register the device's port amd the B register the number of bytes to write.

The instructions involved are **OUTD**[13] and **OUTI**[14] with their associated repeat instructions **OTDR**[15] and **OTIR**[16][17].

There is, however, one big difference, and that is the address bus.

21.4.1 Address Bus

When selecting the output port, just like the input instructions, the B and C registers are placed on the address bus, with C being placed on bits $A_0 \ldots A_7$ of the address bus whilst the value of the B register is placed on bits $A_8 \ldots A_{15}$.

However, for the output instructions, the B register is decremented[2] before placing the result onto the address bus.

This is opposite to how the input[18] instructions use the address bus as they decrement the B register after reading from the bus.

If B is set to 0 before one of these instructions then 256 bytes will be written to the device.

13: **OUTD** Section 22.41 on page 343

14: **OUTI** Section 22.42 on page 345

15: **OTDR** Section 22.38 on page 339

16: **OTIR** Section 22.39 on page 340

17: **OTDR** and **OTIR** are so named instead of OUTDR or OUTIR as Z80 assemblers use a maximum of 4 characters for the name

[2] *Z80 CPU User Manual*, (Zilog Inc Aug. 2016) PP.309-316

18: input Subsection 21.3.1 on page 273

Z80 Instruction Set 22

This chapter describes every single instruction implemented by the Z80 microprocessor.

Each section describes an individual instruction, listing the address modes supported, which processors have that instruction and the processor flags that are changed by it, or affects the operation of the instruction.

Within this section, the following nomenclature applies:

Z80 nomenclature

Operand	Target	Details
a	8-bit register	one of: a, b, c, d, e, h or l
bc	16-bit register	one of: bc, de or hl
ix	16-bit index register	one of ix or iy
N	8-bit number	0 ... 255
NN	16-bit number or address	0 ... 65535
+d	8-bit offset	used with ix or iy instructions
DIS	8-bit displacement	used with relative jumps
af	Accumulator and Flags	in one operation
	e.g. `PUSH af`	Section 22.44 on page 348
af'	Alternate register	
	e.g. `EX af,af'`	Section 22.18 on page 302

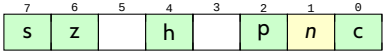
22.1 ADC
Add with Carry

Function $A \leftarrow A + s + P_c$

$HL \leftarrow HL + s + C$

Description Add Accumulator or HL with the specified operand along with carry

Flags



s Most significant bit of result is set

z Result was zero

h Carry from bit 3 to 4

p Parity of result is even

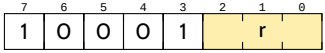
c Operation caused bit to be carried over

n Cleared

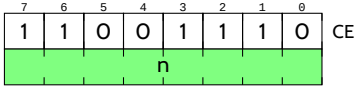
Legend	
d	Data Offset
n	Immediate Data
r	Register
111	A
000	B
001	C
010	D
011	E
100	H
101	L
s	Register Pair
00	BC
01	DE
10	HL
11	SP

Table 22.1: Z80 Instruction Format ADC

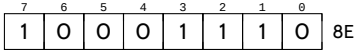
ADC A,r



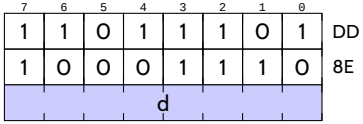
ADC A,n



ADC A,(HL)



ADC A,(IX+d)



ADC A,(IY+d)

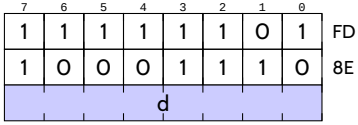


Table 22.1: Z80 Instruction Format ADC (continued)

	7	6	5	4	3	2	1	0	
ADC HL,ss	1	1	1	0	1	1	0	1	ED
	0	1	s		1	0	1	0	

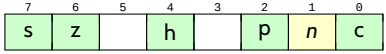
22.2 ADD
Add without Carry

Function $A \leftarrow A + s$

$HL \leftarrow HL + s$

Description The operand is added to the destination, with the result stored in the destination.

Flags



s Most significant bit of result is set

z Result was zero

h Carry from bit 3 to 4

p Parity of result is even

c Operation caused bit to be carried over

n Cleared

Table 22.2: Z80 Instruction ADD

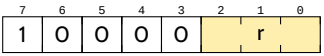
Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`ADD a,a`	`87`	1	1	4		
`ADD a,b`	`80`	1	1	4		
`ADD a,c`	`81`	1	1	4		
`ADD a,d`	`82`	1	1	4		
`ADD a,e`	`83`	1	1	4		
`ADD a,h`	`84`	1	1	4		
`ADD a,l`	`85`	1	1	4		
`ADD a,N`	`C6`	2	2	7		
`ADD a,(hl)`	`86`	1	2	7		
`ADD a,(ix+d)`	`DD86nn`	3	5	19		
`ADD a,(iy+d)`	`FD86nn`	3	5	19		
`ADD hl,bc`	`09`	1	3	11		
`ADD hl,de`	`19`	1	3	11		

Table 22.2: Z80 Instruction ADD (continued)

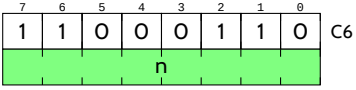
Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`ADD hl,hl`	`29`	1	3	11		
`ADD hl,sp`	`39`	1	3	11		
`ADD ix,bc`	`DD09`	2	4	15		
`ADD ix,de`	`DD19`	2	4	15		
`ADD ix,ix`	`DD29`	2	4	15		
`ADD ix,sp`	`DD39`	2	4	15		
`ADD iy,bc`	`FD09`	2	4	15		
`ADD iy,de`	`FD19`	2	4	15		
`ADD iy,iy`	`FD29`	2	4	15		
`ADD iy,sp`	`FD39`	2	4	15		

Table 22.3: Z80 Instruction Format ADD

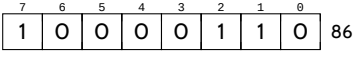
ADD A,r



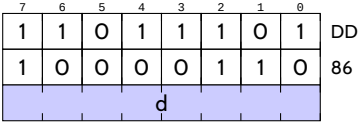
ADD A,n



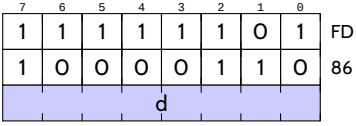
ADD A,(HL)



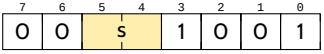
ADD A,(IX+d)



ADD A,(IY+d)



ADD HL,ss



Legend

d	Data Offset
n	Immediate Data
r	Register

111	A
000	B
001	C
010	D
011	E
100	H
101	L

s	Register Pair
00	BC
01	DE
10	HL
11	SP

22.3 AND
Logical AND

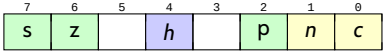
Figure 22.1: AND truth table

Memory \ Accumulator	0	1
0	0	0
1	0	1

Function $A \leftarrow A\ s$

Description The accumulator and the operand are logically and'ed, with the result stored in the destination.

Flags



s Most significant bit of result is set

z Result was zero

h Set

p Parity of result is even

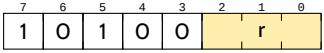
c Cleared

n Cleared

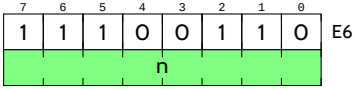
Legend	
d	Data Offset
n	Immediate Data
r	Register
111	A
000	B
001	C
010	D
011	E
100	H
101	L

Table 22.4: Z80 Instruction Format AND

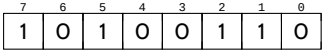
AND r



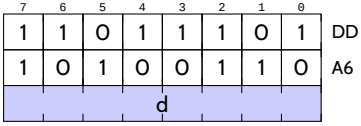
AND n



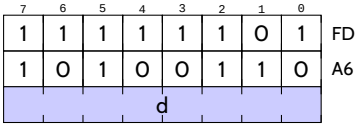
AND (HL)



AND (IX+d)



AND (IY+d)



22.4 BIT
Test bit value

Function $P_z \leftarrow \overline{r_b}$

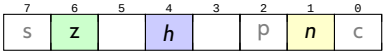
$P_z \leftarrow \overline{(HL)_b}$

$P_z \leftarrow \overline{(IX+d)_b}$

$P_z \leftarrow \overline{(IY+d)_b}$

Description The specified bit of the memory location addressed by the contents of the HL register is tested and the Z flag is set according to the result.

Flags



z Value of the bit

h Set

n Cleared

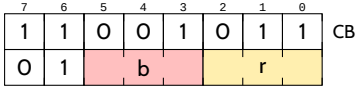
See also **RES**[1], **SET**[2]

1: **RES** Section 22.45 on page 350

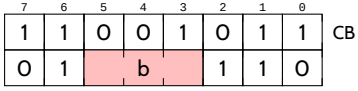
2: **SET** Section 22.62 on page 378

Table 22.5: Z80 Instruction Format BIT

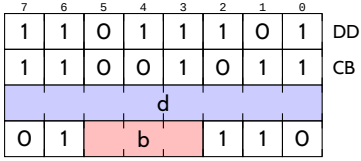
BIT b,r



BIT b,(HL)



BIT b,(IX+d)



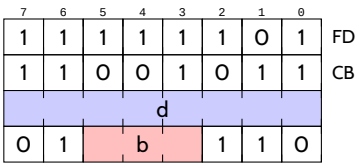
Legend

b	Bit Number
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

d	Data Offset
r	Register
111	A
000	B
001	C
010	D
011	E
100	H
101	L

Table 22.5: Z80 Instruction Format BIT (continued)

BIT b,(IY+d)



22.5 CALL
Call Subroutine

Function $(SP - 1) \leftarrow PC_{high}$

$(SP - 2) \leftarrow PC_{low}$

$SP \leftarrow SP - 2$

$PC_{high} \leftarrow 0$

$PC_{low} \leftarrow p$

Description Calls a subroutine at the specified address. blabk line

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

See also **RET**[3], **RST**[4]

3: **RET** Section 22.46 on page 354

4: **RST** Section 22.59 on page 371

Table 22.6: Z80 Instruction CALL

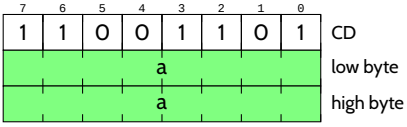
Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`CALL c,NN`	`DC`	3	5	17	3	10
`CALL m,NN`	`FC`	3	5	17	3	10
`CALL nc,NN`	`D4`	3	5	17	3	10
`CALL nz,NN`	`C4`	3	5	17	3	10
`CALL pe,NN`	`EC`	3	5	17	3	10
`CALL po,NN`	`E4`	3	5	17	3	10
`CALL p,NN`	`F4`	3	5	17	3	10
`CALL z,NN`	`CC`	3	5	17	3	10
`CALL NN`	`CD`	3	5	17		

Legend

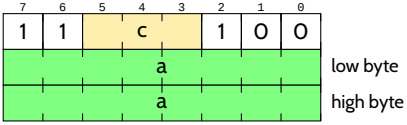
a	Address
c	Condition
`000`	NZ
`001`	Z
`010`	NC
`011`	C
`100`	PO
`101`	PE
`110`	P
`111`	M

Table 22.7: Z80 Instruction Format CALL

CALL nn



CALL cc,nn

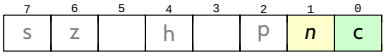


22.6 CCF
Complement Carry Flag

Function $C \leftarrow \overline{C}$

Description The carry flag is complemented.

Flags



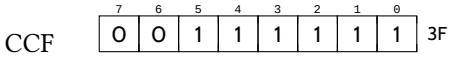
n Cleared

c Complement of original value

Table 22.8: Z80 Instruction CCF

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
CCF	3F	1	1	4		

Table 22.9: Z80 Instruction Format CCF

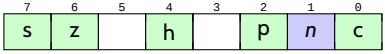


22.7 CP
Compare to Accumulator

Function $A - s$

Description Subtract the operand from the accumulator and set the flags based on the result. The result of the subtraction is discarded.

Flags



s Most significant bit of result is set

z Result was zero

h Carry from bit 3 to 4

p Parity of result is even

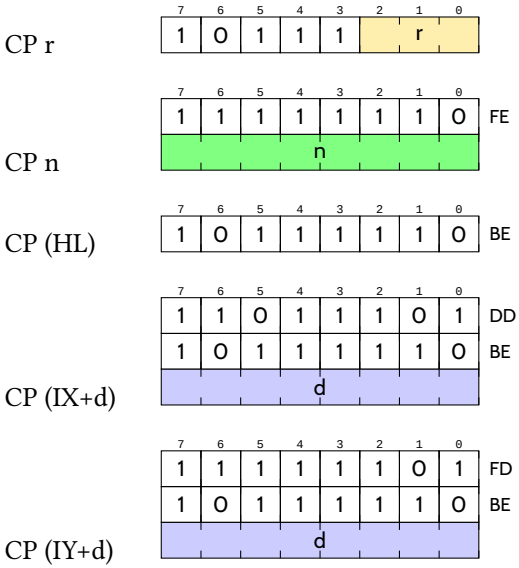
c Operation caused bit to be carried over

n Set

Table 22.10: Z80 Instruction CP

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`CP a`	`BF`	1	1	4		
`CP b`	`B8`	1	1	4		
`CP c`	`B9`	1	1	4		
`CP d`	`BA`	1	1	4		
`CP e`	`BB`	1	1	4		
`CP h`	`BC`	1	1	4		
`CP l`	`BD`	1	1	4		
`CP N`	`FE`	2	2	7		
`CP (hl)`	`BE`	1	2	7		
`CP (ix+d)`	`DDBEnn`	3	5	19		
`CP (iy+d)`	`FDBEnn`	3	5	19		

Table 22.11: Z80 Instruction Format CP



Legend	
d	Data Offset
n	Immediate Data
r	Register

111	A
000	B
001	C
010	D
011	E
100	H
101	L

22.8 CPD
Compare with decrement

Function $A - (HL)$

$HL \leftarrow HL - 1$

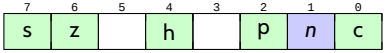
$BC \leftarrow BC - 1$

Description Subtract the contents of the memory location addressed by HL from the accumulator but discard the results.

The result however affects the flags.

Post op Decrement both the HL and BC registers after the comparison.

Flags



s Most significant bit of result is set

h Carry from bit 3 to 4

p Parity of result is even

c Operation caused bit to be carried over

n Set

z Set is $A = (HL)$

p Reset if $BC = 0$ after execution, otherwise Set.

See also **CPDR**[5], **CPI**[6], **CPIR**[7]

5: **CPDR** Section 22.9 on page 290

6: **CPI** Section 22.10 on page 291

7: **CPIR** Section 22.11 on page 292

22.8.1 Compare Block Instructions

Section 21.2 on page 272 covers this in more detail.

There are four instructions which compare against a block of memory which are listed below. The `CPI*` instructions increment the address so start from the beginning of the block whilst the `CPD*` instructions decrement the address so start from the end.

The `CP*R` instructions repeat until either a match occurs or $BC = 0$.

The timings for those are for each iteration with the not taken timings for when a match or the end is reached.

The total timings will be dependent on the number of iterations taken.

Table 22.12: Z80 Instruction CPD

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
CPD	EDA9	2	4	16		

Table 22.13: Z80 Instruction Format CP__

	7	6	5	4	3	2	1	0	
	1	1	1	0	1	1	0	1	ED
CP_F	1	0	1	C	D	0	0	1	

Legend	
C	Repeat Instruction
0	run once
1	while $BC \neq 0$
D	Direction
0	Increment
1	Decrement

22.9 CPDR
Block compare with decrement

The Opcodes and timings are shown in Subsection 22.8.1

Function $A - (HL)$

$HL \leftarrow HL - 1$

$BC \leftarrow BC - 1$

Description Subtract the contents of the memory location addressed by HL from the accumulator but discard the results.

The result however affects the flags.

Post op Decrement both the HL and BC registers after the comparison.

Repeat If $BC \neq 0$ *and* $A \neq (HL)$ then the program counter is decremented by two and the instruction is re-executed.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	*n*	c

s Most significant bit of result is set

h Carry from bit 3 to 4

p Parity of result is even

c Operation caused bit to be carried over

n Set

z Set is $A = (HL)$

p Reset if $BC = 0$ after execution, otherwise Set.

See also **CPD**[8], **CPI**[9], **CPIR**[10]

This instruction is usually used to find the last[11] byte in a block of memory that equals the accumulator.

8: **CPD** Section 22.8 on page 288

9: **CPI** Section 22.10 on page 291

10: **CPIR** Section 22.11 on page 292

11: Last as it decrements so searches from the end of the block

22.10 CPI
Compare with increment

The Opcodes and timings are shown in Subsection 22.8.1

Function $A - (HL)$

$HL \leftarrow HL + 1$

$BC \leftarrow BC - 1$

Description Subtract the contents of the memory location addressed by HL from the accumulator but discard the results.

The result however affects the flags.

Post op Increment the HL register and decrement the BC register after the comparison.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	*n*	c

s Most significant bit of result is set

h Carry from bit 3 to 4

p Parity of result is even

c Operation caused bit to be carried over

n Set

z Set is $A = (HL)$

p Reset if $BC = 0$ after execution, otherwise Set.

See also **CPD**[12], **CPDR**[13], **CPIR**[14]

12: **CPD** Section 22.8 on page 288

13: **CPDR** Section 22.9 on page 290

14: **CPIR** Section 22.11 on page 292

22.11 CPIR
Block compare with increment

The Opcodes and timings are shown in Subsection 22.8.1

Function $A - (HL)$

$HL \leftarrow HL + 1$

$BC \leftarrow BC - 1$

Description Subtract the contents of the memory location addressed by HL from the accumulator but discard the results.

The result however affects the flags.

Post op Increment the HL register and decrement the BC register after the comparison.

Repeat If $BC \neq 0$ *and* $A \neq (HL)$ then the program counter is decremented by two and the instruction is re-executed.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	*n*	c

s Most significant bit of result is set

h Carry from bit 3 to 4

p Parity of result is even

c Operation caused bit to be carried over

n Set

z Set is $A = (HL)$

p Reset if $BC = 0$ after execution, otherwise Set.

See also **CPD**[15], **CPDR**[16], **CPI**[17]

This instruction is usually used to find the first[18] byte in a block of memory that equals the accumulator.

15: **CPD** Section 22.8 on page 288

16: **CPDR** Section 22.9 on page 290

17: **CPI** Section 22.10 on page 291

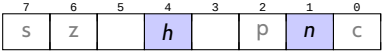
18: First as it increments so searches from the start of the block

22.12 CPL Complement Accumulator

Function $A \leftarrow \overline{A}$

Description The contents of the accumulator is inverted, or complemented, with the result stored in the accumulator.

Flags



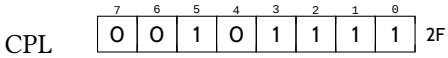
h Set

n Set

Table 22.14: Z80 Instruction CPL

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
CPL	2F	1	1	4		

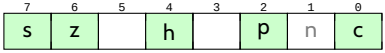
Table 22.15: Z80 Instruction Format CPL



22.13 DAA
Decimal Adjust Accumulator

Description Conditionally add 6 to either of the left or right nibbles of the accumulator, based on status of the flags after arithmetic operations to convert the result toSubtract the operand from the accumulator and set the flags based on the result. The result of the subtraction is discarded.

Flags



s Most significant bit of result is set

z Result was zero

h Carry from bit 3 to 4

p Parity of result is even

c Operation caused bit to be carried over

Table 22.16: Z80 Instruction DAA

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
DAA	27	1	1	4		

Table 22.17: Z80 Instruction Format DAA

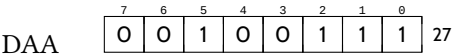


Table 22.18: Adjustments to Accumulator performed by the DAA instruction

Before execution					Added	After execution
N	C	high nibble	H	low nibble	to A	C
0	0	0x0 ...0x9	0	0x0 ...0x9	0x00	0
		0x0 ...0x8		0xA ...0xF	0x06	
		0x0 ...0x9	1	0x0 ...0x3	0x06	
		0xA ...0xF	0	0x0 ...0x9	0x60	1
		0x9 ...0xF		0xA ...0xF	0x66	
		0xA ...0xF	1	0x0 ...0x3	0x66	
	1	0x0 ...0x2	0	0x0 ...0x9	0x60	
		0x0 ...0x2		0xA ...0xF	0x66	
		0xA ...0x3	1	0x0 ...0x3	0x66	
1	0	0x0 ...0x9	0	0x0 ...0x9	0x00	0
		0x0 ...0x8	1	0x6 ...0xF	0xFA	
	1	0x7 ...0xF	0	0x0 ...0x9	0xA0	1
		0x6 ...0xF	1	0x6 ...0xF	0x9A	

N is clear following **ADD**, **ADC** or **INC**
N is set following **SUB**, **SBC**, **DEC** or **NEG**

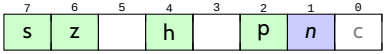
22.14 DEC
Decrement register or memory

Function $m \leftarrow m - 1$

Description Decrement the operand by one and store the result back at the same location.

8-bit registers and memory (HL)

Flags



s Most significant bit of result is set

z Result was zero

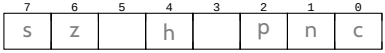
h Carry from bit 3 to 4

p Parity of result is even

n Set

16-bit register pairs, IX and IY

Flags



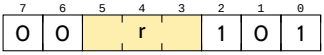
Decrementing BC, DE, HL, IX or IY has no effect on the flags

See also **INC**[19]

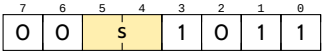
19: **INC** Section 22.23 on page 310

Table 22.19: Z80 Instruction Format DEC

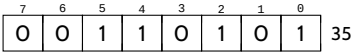
DEC r



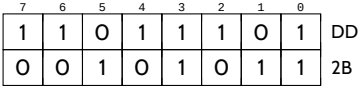
DEC ss



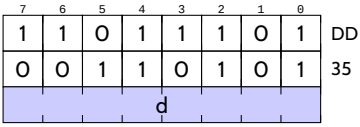
DEC (HL)



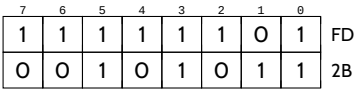
DEC IX



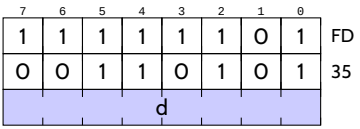
DEC (IX+d)



DEC IY



DEC (IY+d)



Legend

d	Data Offset
r	Register

111	A
000	B
001	C
010	D
011	E
100	H
101	L

s	Register Pair
00	BC
01	DE
10	HL
11	SP

22.15 DI
Disable Maskable Interrupts

Function $IFF \leftarrow 0$

Description Reset the Interrupt Flip Flops, disabling all maskable interrupts. They can be re-enabled with **EI** instruction.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

See also **EI**[20]

20: **EI** Section 22.17 on page 301

Table 22.20: Z80 Instruction DI

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
DI	F3	1	1	4		

Table 22.21: Z80 Instruction Format DI

	7	6	5	4	3	2	1	0
DI	1	1	1	1	0	0	1	1

22.16 DJNZ Decrement B and jump on non-zero

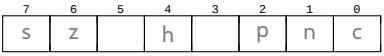
Function $B \leftarrow B - 1$

$PC \leftarrow PC + e if B \neq 0$

Description Decrements the B register. If the result is not zero then the offset is added to the program counter using two's complement arithmetic.

The destination address must be within $-126 \cdots +129$ bytes from the instruction's address.

Flags

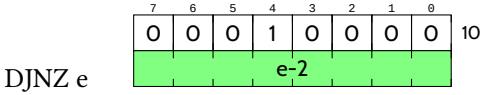


This instruction enables performing loops with only two bytes of memory used for the instruction. Listing 22.1 on page 300 shows both **DJNZ** in use along side equivalent routines which are a byte longer and take longer to execute.

Table 22.22: Z80 Instruction DJNZ

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
DJNZ DIS	10	2	3	13	2	8

Table 22.23: Z80 Instruction Format DJNZ



Legend	
e	Relative Address

Listing 22.1: Example of DJNZ

```
        LD B, 10
.loop
        DJNZ loop
        RET

        LD B, 10
.loop
        DEC B
        JR NZ,loop
        RET

        LD B, 10
.loop
        DEC B
        RET Z
        JR loop
```

22.17 EI
Enable Maskable Interrupts

Function $IFF \leftarrow 1$

Description Set the Interrupt Flip Flops, enabling all maskable interrupts. They can be disabled with **DI** instruction.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

See also **DI**[21]

21: **DI** Section 22.15 on page 298

Table 22.24: Z80 Instruction EI

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
EI	FB	1	1	4		

Table 22.25: Z80 Instruction Format EI

	7	6	5	4	3	2	1	0
EI	1	1	1	1	1	0	1	1

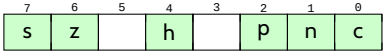
22.18 EX
Exchange Registers

Function $AF \leftrightarrow AF'$

Description Exchange a pair of register pairs.

AF,AF' Exchange the accumulator and flags with their alternate equivalents.

Flags



s Most significant bit of result is set

z Result was zero

h Carry from bit 3 to 4

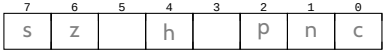
p Parity of result is even

n BCD Subtract

c Operation caused bit to be carried over

DE,HL Exchange the DE and HL register pairs.

Flags



(SP),HL Exchange the HL register pair with the top 2 bytes on the stack.

(SP),IX Exchange the IX register with the top 2 bytes on the stack.

(SP),IY Exchange the IY register with the top 2 bytes on the stack.

Flags

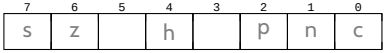


Table 22.26: Z80 Instruction EX

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`EX af,af'`	`08`	1	1	4		
`EX de,hl`	`EB`	1	1	4		
`EX (sp),hl`	`E3`	1	5	19		
`EX (sp),ix`	`DDE3`	2	6	23		
`EX (sp),iy`	`FDE3`	2	6	23		

Table 22.27: Z80 Instruction Format EX

	7	6	5	4	3	2	1	0	
EX AF,AF'	0	0	0	0	1	0	0	0	08
EX DE,HL	1	1	1	0	1	0	1	1	EB
EX (SP),HL	1	1	1	0	1	1	0	0	E3
EX (SP),IX	1	1	0	1	1	1	0	1	DD
	1	1	1	0	1	1	0	0	E3
EX (SP),IY	1	1	1	1	1	1	0	1	FD
	1	1	1	0	1	1	0	0	E3

22.19 EXX
Exchange Alternate Registers

Function $BC \leftrightarrow BC'$

$DE \leftrightarrow DE'$

$HL \leftrightarrow HL'$

Description Exchange the BC, DE and HL registers with their alternates.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

Table 22.28: Z80 Instruction EXX

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
EXX	D9	1	1	4		

Table 22.29: Z80 Instruction Format EXX

	7	6	5	4	3	2	1	0	
EXX	1	1	0	1	1	0	0	1	D9

22.20 HALT
Halt the CPU

Description Suspends the CPU so that it continually executes **NOP**[22] until an interrupt or reset is received.

22: **NOP** Section 22.36 on page 336

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

Table 22.30: Z80 Instruction HALT

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
HALT	76	1	1	4		

Table 22.31: Z80 Instruction Format HALT

HALT

7	6	5	4	3	2	1	0	
0	1	1	1	0	1	1	0	76



The reason it executes **NOP** rather than shutdown is to keep the memory refresh cycles running.

Without those refresh cycles, dynamic ram would be erased in a short period of time.

22.21 IM
Set Interrupt Mode

Description Sets the interrupt mode.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

The Z80 has three modes that interrupts can operate under.

Table 22.32: Z80 Instruction IM

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
IM 0	ED46	2	2	8		
IM 1	ED56	2	2	8		
IM 2	ED5E	2	2	8		

Table 22.33: Z80 Instruction Format IM

	7	6	5	4	3	2	1	0	
IM 0	1	1	1	0	1	1	0	1	ED
	0	1	0	0	0	1	1	0	46
IM 1	1	1	1	0	1	1	0	1	ED
	0	1	0	1	0	1	1	0	56
IM 2	1	1	1	0	1	1	0	1	ED
	0	1	0	1	1	1	1	0	5E

22.21.1 Interrupt Mode 0

In mode 0 an interrupting device may insert one instruction onto the data bus for execution, the first byte of which must occur during the interrupt acknowledge cycle.

22.21.2 Interrupt Mode 1

In mode 1 an **RST**[23] `0x0036` instruction would be executed whenever an interrupt occurs.

23: **RST** Section 22.59 on page 371

22.21.3 Interrupt Mode 2

In mode 2, when an interrupt occurs, one byte of data must be provided by the peripheral which is used as the low order of an address. The high order of this vector address is taken from the contents of the I register. This points to a second address stored in memory, which is loaded into the program counter and begins execution.

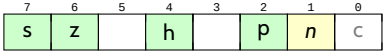
22.22 IN
Load register from Port

Description Load a register from a port.

r,(C)

Address bus The C register provides bits $A_0 \ldots A_7$ of the address bus.
B provides bits $A_8 \ldots A_{15}$.

Flags



s Most significant bit of result is set

z Result was zero

h Carry from bit 3 to 4

p Parity of result is even

n Cleared

A,(N)

Address bus The literal N provides bits $A_0 \ldots A_7$ of the address bus.
The Accumulator provides bits $A_8 \ldots A_{15}$.

Flags

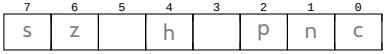


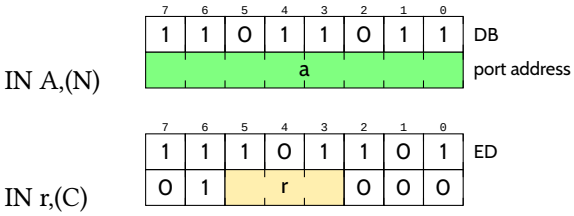
Table 22.34: Z80 Instruction IN

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`IN a,(c)`	`ED78`	2	3	12		
`IN a,(N)`	`DB`	2	3	11		
`IN b,(c)`	`ED40`	2	3	12		
`IN c,(c)`	`ED48`	2	3	12		
`IN d,(c)`	`ED50`	2	3	12		
`IN e,(c)`	`ED58`	2	3	12		
`IN f,(c)`	`ED70`	2	3	12		

Table 22.34: Z80 Instruction IN (continued)

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`IN h,(c)`	`ED60`	2	3	12		
`IN l,(c)`	`ED68`	2	3	12		

Table 22.35: Z80 Instruction Format IN



Legend

a	Address
r	Register
`111`	A
`000`	B
`001`	C
`010`	D
`011`	E
`100`	H
`101`	L

IN F,(C)

rrr above can also have the value 110 which is the F flag register. So there is the possibility of an **IN F,(C)** instruction with code `0xED70`. The usefulness of this instruction is, however, debatable.

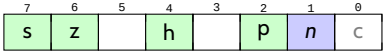
22.23 INC
Increment register or memory

Function $m \leftarrow m + 1$

Description Increment the operand by one and store the result back at the same location.

8-bit registers and memory (HL)

Flags



s Most significant bit of result is set

z Result was zero

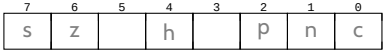
h Carry from bit 3 to 4

p Parity of result is even

n Set

16-bit register pairs, IX and IY

Flags



Decrementing BC, DE, HL, IX or IY has no effect on the flags

See also **DEC**[24]

24: **DEC** Section 22.14 on page 296

Table 22.36: Z80 Instruction Format INC

	7	6	5	4	3	2	1	0	
INC r	0	0	r			1	0	0	
INC ss	0	0	s		0	0	1	1	
INC (HL)	0	0	1	1	0	1	0	0	34
INC IX	1	1	0	1	1	1	0	1	DD
	0	0	1	0	0	0	1	1	23
INC (IX+d)	1	1	0	1	1	1	0	1	DD
	0	0	1	1	0	1	0	0	34
	d								
INC IY	1	1	1	1	1	1	0	1	FD
	0	0	1	0	0	0	1	1	23
INC (IY+d)	1	1	1	1	1	1	0	1	FD
	0	0	1	1	0	1	0	0	34
	d								

Legend

d	Data Offset
r	Register

111	A
000	B
001	C
010	D
011	E
100	H
101	L

s	Register Pair
00	BC
01	DE
10	HL
11	SP

22.24 IND
Input with Decrement

Section 21.3 on page 273 covers this in more detail.

Function $(HL) \leftrightarrow (C)$

$B \leftarrow B - 1$

$HL \leftarrow HL - 1$

Description The peripheral device addressed by the C register is read and the result is loaded into the memory location addressed by the HL register pair.

Address bus The C register provides bits $A_0 \ldots A_7$ of the address bus.
B provides bits $A_8 \ldots A_{15}$.

Post op The B register and the HL register pair are then each decremented.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	*n*	c

z Set if $B = 0$, reset if $B \neq 0$

n Set

See also **INDR**[25], **INI**[26], **INIR**[27]

25: **INDR** Section 22.25 on page 314

26: **INI** Section 22.26 on page 315

27: **INIR** Section 22.27 on page 316

22.24.1 Input Block Instructions

Section 21.3 on page 273 covers this in more detail.

There are four instructions which read from a peripheral device and writes to a block of memory. The `INI*` instructions increment the address so they start from the beginning of the block whilst the `IND*` instructions decrement the address so start from the end.

The `IN*R` instructions will automatically repeat until $B = 0$.

The timings for those are for each iteration with the not taken timings for when the end is reached.

The total timings will be dependent on the number of iterations taken.

Table 22.37: Z80 Instruction IND

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`IND`	EDAA	2	4	16		

Table 22.38: Z80 Instruction Format IN__

	7	6	5	4	3	2	1	0	
	1	1	1	0	1	1	0	1	ED
IN␣␣	1	0	1	C	D	0	1	0	

Legend	
C	Repeat Instruction
0	run once
1	while $BC \neq 0$
D	Direction
0	Increment
1	Decrement

22.25 INDR
Block input with Decrement

Section 21.3 on page 273 covers this in more detail.

The Opcodes and timings are shown in Subsection 22.24.1 on the previous page

Function $(HL) \leftrightarrow (C)$

$B \leftarrow B - 1$

$HL \leftarrow HL - 1$

Description The peripheral device addressed by the C register is read and the result is loaded into the memory location addressed by the HL register pair.

Address bus The C register provides bits $A_0 \ldots A_7$ of the address bus.
B provides bits $A_8 \ldots A_{15}$.

Post op The B register and the HL register pair are then each decremented.

Repeat If $B \neq 0$ then the program counter is decremented by 2 and the instruction is re-executed.

Flags

7	6	5	4	3	2	1	0
s	*z*		h		p	*n*	c

z Set

n Set

See also **IND**[28], **INI**[29], **INIR**[30]

28: IND Section 22.24 on page 312

29: INI Section 22.26 on page 315

30: INIR Section 22.27 on page 316

22.26 INI
Input with Increment

Section 21.3 on page 273 covers this in more detail.

The Opcodes and timings are shown in Subsection 22.24.1 on the previous page

Function $(HL) \leftrightarrow (C)$

$B \leftarrow B - 1$

$HL \leftarrow HL + 1$

Description The peripheral device addressed by the C register is read and the result is loaded into the memory location addressed by the HL register pair.

Address bus The C register provides bits $A_0 \ldots A_7$ of the address bus.
B provides bits $A_8 \ldots A_{15}$.

Post op The B register is decremented and the HL register pair is incremented.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	*n*	c

z Set if $B = 0$, reset if $B \neq 0$

n Set

See also **IND**[31], **INDR**[32], **INIR**[33]

31: IND Section 22.24 on page 312

32: INDR Section 22.25 on page 314

33: INIR Section 22.27 on page 316

22.27 INIR
Block input with Increment

Section 21.3 on page 273 covers this in more detail.

The Opcodes and timings are shown in Subsection 22.24.1 on the previous page

Function $(HL) \leftrightarrow (C)$

$B \leftarrow B - 1$

$HL \leftarrow HL + 1$

Description The peripheral device addressed by the C register is read and the result is loaded into the memory location addressed by the HL register pair.

Address bus The C register provides bits $A_0 \ldots A_7$ of the address bus.
B provides bits $A_8 \ldots A_{15}$.

Post op The B register is decremented and the HL register pair is incremented.

Repeat If $B \neq 0$ then the program counter is decremented by 2 and the instruction is re-executed.

Flags

7	6	5	4	3	2	1	0
s	*z*		h		p	*n*	c

z Set

n Set

See also **IND**[34], **INI**[35], **INIR**[36]

34: IND Section 22.24 on page 312
35: INI Section 22.26 on page 315
36: INIR Section 22.27 on page 316

22.28 JP
Jump to location

Description Set the program counter to the provided address, causing the processor to jump to the new address.

jp NN Use the provided address as the destination.

jp cc,NN Perform the jump only if the specified condition is true.

jp (ss) Use the address stored in the location pointed to by the register pair as the new address.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

See also DJNZ[37], JR[38]

37: DJNZ Section 22.16 on page 299

38: JR Section 22.29 on page 319

Table 22.39: Z80 Instruction JP

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
JP c,NN	DA	3	3	10		
JP m,NN	FA	3	3	10		
JP nc,NN	D2	3	3	10		
JP nz,NN	C2	3	3	10		
JP pe,NN	EA	3	3	10		
JP po,NN	E2	3	3	10		
JP p,NN	F2	3	3	10		
JP z,NN	CA	3	3	10		
JP NN	C3	3	3	10		
JP (hl)	E9	1	1	4		
JP (ix)	DDE9	2	2	8		
JP (iy)	FDE9	2	2	8		

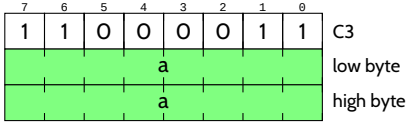
Legend

a	Address
c	Condition

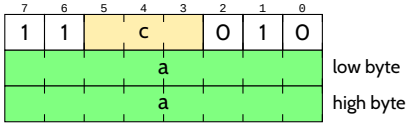
000	NZ
001	Z
010	NC
011	C
100	PO
101	PE
110	P
111	M

Table 22.40: Z80 Instruction Format JP

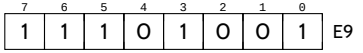
JP nn



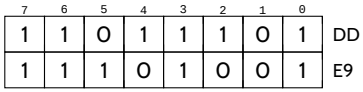
JP cc,nn



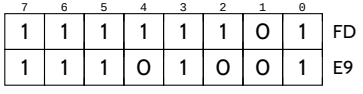
JP (HL)



JP (IX)



JP (IY)



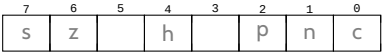
22.29 JR
Jump to relative location

Description Jump to a location a set distance[39] away from the current address

jr e Add the offset in two's complement to the program counter, causing the processor to jump to the new address.

jr cc,e If the specified condition is true then add the offset in two's complement to the program counter, causing the processor to jump to the new address.

Flags



See also **DJNZ**[40], **JP**[41]

39: The offset stored is 2 less than the actual distance

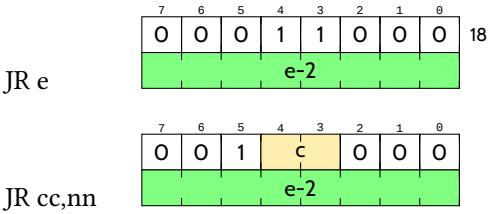
40: **DJNZ** Section 22.16 on page 299

41: **JP** Section 22.28 on page 317

Table 22.41: Z80 Instruction JR

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
JR c,DIS	38	2	3	12	2	6
JR nc,DIS	30	2	3	12	2	6
JR nz,DIS	20	2	3	12	2	6
JR z,DIS	28	2	3	12	2	6
JR DIS	18	2	3	12		

Table 22.42: Z80 Instruction Format JR



Legend	
c	Condition
00	NZ
01	Z
10	NC
11	C
e	Relative Address

22.30 LD
Load register or memory

Function Load the value from a source register or memory location and store it in a register or memory location.

ld r,n

Description Set the value of a register.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

For register pairs, this would use two bytes, the low byte at the address and the high byte at the next higher address.

ld r,(nn)

Description Store the value of a memory location in a register.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

ld (nn),r

Description Store the value of a register in a memory location.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

ld r,(rr)

Description Store the value of a memory location pointed to by a register pair in a register.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

ld (rr),r

Description Store the value of a register in a memory location pointed to by a register pair.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

ld (rr),n

Description Store a numeric value in a memory location pointed to by a register pair.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

ld r,r'

Description Set register `r` with the value of `r'`.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

ld a,I

Description Set accumulator to the interrupt vector register I z80 flag

ld I,a

Description Set interrupt vector register I to the accumulator. z80 flag

ld a,R

Description Set accumulator to the Memory Refresh register R z80 flag

ld R,a

Description Set the Memory Refresh register R to the accumulator. z80 flag

Errata

In[3] the **LD r,(HL)** instructions are listed separately from the rest of the **LD** instructions which can cause confusion.

[3] *Programming the Z80*, Zaks (Sybex 1982) P.356

Table 22.43: Z80 Instruction LD

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
LD a,a	7F	1	1	4		
LD a,b	78	1	1	4		
LD a,c	79	1	1	4		
LD a,d	7A	1	1	4		
LD a,e	7B	1	1	4		
LD a,h	7C	1	1	4		
LD a,i	ED57	2	2	9		
LD a,l	7D	1	1	4		
LD a,r	ED5F	2	2	9		
LD a,N	3E	1	2	7		
LD a,(bc)	0A	1	2	7		
LD a,(de)	1A	1	2	7		
LD a,(hl)	7E	1	2	7		
LD a,(ix+d)	DD7Enn	3	5	19		
LD a,(ix+d)	FD7Enn	3	5	19		
LD a,(NN)	3A	3	4	13		
LD bc,NN	01	3	3	10		
LD bc,(NN)	ED4B	2	6	20		
LD b,a	47	1	1	4		
LD b,b	40	1	1	4		
LD b,c	41	1	1	4		
LD b,d	42	1	1	4		
LD b,e	43	1	1	4		
LD b,h	44	1	1	4		
LD b,l	45	1	1	4		
LD b,n	06	1	2	7		
LD b,(hl)	46	1	2	7		
LD b,(ix+d)	DD46nn	3	5	19		
LD b,(iy+d)	FD46nn	3	5	19		
LD c,a	4F	1	1	4		
LD c,b	48	1	1	4		
LD c,c	49	1	1	4		
LD c,d	4A	1	1	4		
LD c,e	4B	1	1	4		

Table 22.43: Z80 Instruction LD (continued)

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
LD c,h	4C	1	1	4		
LD c,l	4D	1	1	4		
LD c,N	0E	1	2	7		
LD c,(hl)	4E	1	2	7		
LD c,(ix+d)	DD4Enn	3	5	19		
LD c,(iy+d)	FD4Enn	3	5	19		
LD de,NN	11	3	3	10		
LD de,(NN)	ED5Bnnnn	4	6	20		
LD d,a	57	1	1	4		
LD d,b	50	1	1	4		
LD d,c	51	1	1	4		
LD d,d	52	1	1	4		
LD d,e	53	1	1	4		
LD d,h	54	1	1	4		
LD d,l	55	1	1	4		
LD d,n	16	1	2	7		
LD d,(hl)	56	1	2	7		
LD d,(ix+d)	DD56nn	3	5	19		
LD d,(iy+d)	FD56nn	3	5	19		
LD e,a	5F	1	1	4		
LD e,b	58	1	1	4		
LD e,c	59	1	1	4		
LD e,d	5A	1	1	4		
LD e,e	5B	1	1	4		
LD e,h	5C	1	1	4		
LD e,l	5D	1	1	4		
LD e,N	1E	1	2	7		
LD e,(hl)	5E	1	2	7		
LD e,(ix+d)	DD5Enn	3	5	19		
LD e,(iy+d)	FD5Enn	3	5	19		
LD hl,NN	21	3	3	10		
LD hl,(NN)	2A	3	5	16		
LD hl,(NN)	ED6Bnnnn	4	6	20		
LD h,a	67	1	1	4		

Table 22.43: Z80 Instruction LD (continued)

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
LD h,b	60	1	1	4		
LD h,c	61	1	1	4		
LD h,d	62	1	1	4		
LD h,e	63	1	1	4		
LD h,h	64	1	1	4		
LD h,l	65	1	1	4		
LD h,n	26	1	2	7		
LD h,(hl)	66	1	2	7		
LD h,(ix+d)	DD66nn	3	5	19		
LD h,(iy+d)	FD66nn	3	5	19		
LD ix,n	DD21nnnn	4	4	14		
LD ix,(NN)	DD2Annnn	4	6	20		
LD iy,n	FD21nnnn	4	4	14		
LD iy,(NN)	FD2Annnn	4	6	20		
LD i,a	ED47	2	2	9		
LD l,a	6F	1	1	4		
LD l,b	68	1	1	4		
LD l,c	69	1	1	4		
LD l,d	6A	1	1	4		
LD l,e	6B	1	1	4		
LD l,h	6C	1	1	4		
LD l,l	6D	1	1	4		
LD l,N	2E	1	2	7		
LD l,(hl)	6E	1	2	7		
LD l,(ix+d)	DD6Enn	3	5	19		
LD l,(iy+d)	FD6Enn	3	5	19		
LD r,a	ED4F	2	2	9		
LD sp,hl	F9	1	1	6		
LD sp,ix	DDF9	2	2	10		
LD sp,iy	FDF9	2	2	10		
LD sp,NN	31	3	3	10		
LD sp,(NN)	ED7Bnnnn	4	6	20		
LD (bc),a	02	1	2	7		
LD (de),a	12	1	2	7		

Table 22.43: Z80 Instruction LD (continued)

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
LD (hl),a	77	1	2	7		
LD (hl),b	70	1	2	7		
LD (hl),c	71	1	2	7		
LD (hl),d	72	1	2	7		
LD (hl),e	73	1	2	7		
LD (hl),h	74	1	2	7		
LD (hl),l	75	1	2	7		
LD (hl),n	36	2	3	10		
LD (ix+d),a	DD77nn	3	5	19		
LD (ix+d),b	DD70nn	3	5	19		
LD (ix+d),c	DD71nn	3	5	19		
LD (ix+d),d	DD72nn	3	5	19		
LD (ix+d),e	DD73nn	3	5	19		
LD (ix+d),h	DD74nn	3	5	19		
LD (ix+d),l	DD75nn	3	5	19		
LD (ix+d),n	DD36nnnn	4	5	19		
LD (iy+d),a	FD77nn	3	5	19		
LD (iy+d),b	FD70nn	3	5	19		
LD (iy+d),c	FD71nn	3	5	19		
LD (iy+d),d	FD72nn	3	5	19		
LD (iy+d),e	FD73nn	3	5	19		
LD (iy+d),h	FD74nn	3	5	19		
LD (iy+d),l	FD75nn	3	5	19		
LD (iy+d),n	FD36nnnn	4	5	19		
LD (NN),a	32	3	4	13		
LD (NN),bc	ED43nnnn	4	6	20		
LD (NN),de	ED53nnnn	4	6	20		
LD (NN),hl	22	3	5	16		
LD (NN),hl	ED63nnnn	4	6	20		
LD (NN),ix	DD22nnnn	4	6	20		
LD (NN),iy	FD22nnnn	4	6	20		
LD (NN),sp	ED73nnnn	4	6	20		

Legend	
d	Data Offset
n	Immediate Data
r	Register

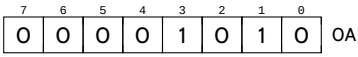
111	A
000	B
001	C
010	D
011	E
100	H
101	L

r'	Register
111	A
000	B
001	C
010	D
011	E
100	H
101	L

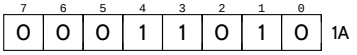
s	Register Pair
00	BC
01	DE
10	HL
11	SP

Table 22.44: Z80 Instruction Format LD

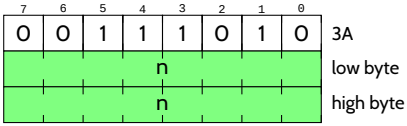
LD A,(BC)



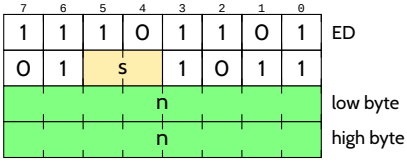
LD A,(DE)



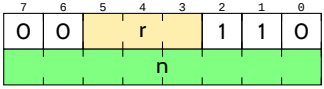
LD A,(nn)



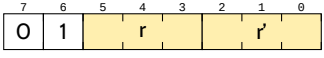
LD ss,(nn)



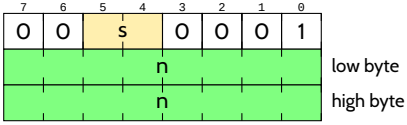
LD r,n



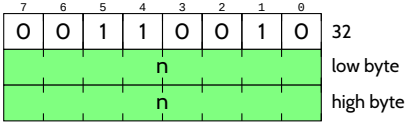
LD r,r'



LD ss,nn



LD (nn),A



LD (nn),HL

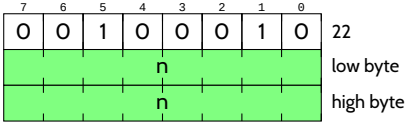
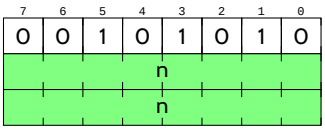
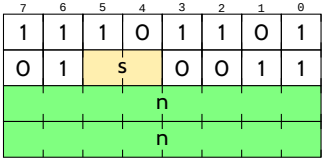


Table 22.44: Z80 Instruction Format LD (continued)

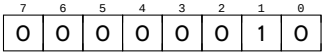
LD HL,(nn)



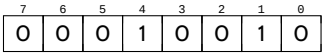
LD (nn),ss



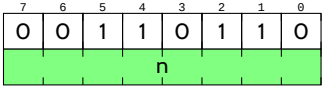
LD (BC),A



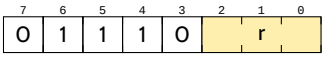
LD (DE),A



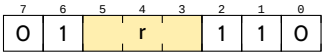
LD (HL),n



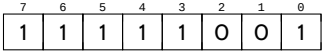
LD (HL),r



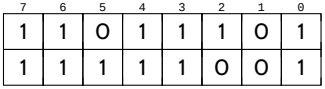
LD r,(HL)



LD SP,HL



LD SP,IX



LD SP,IY

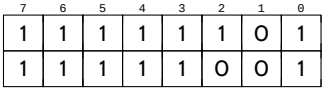
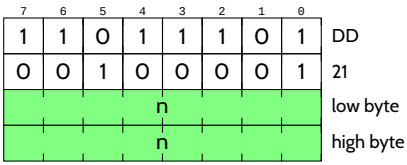
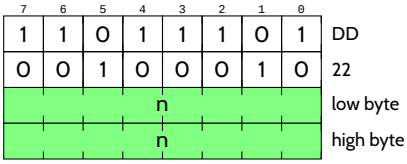


Table 22.44: Z80 Instruction Format LD (continued)

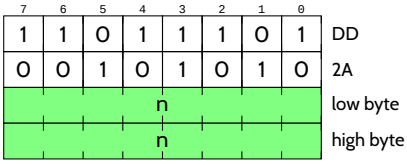
LD IX,n



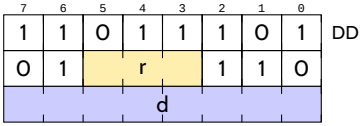
LD (nn),IX



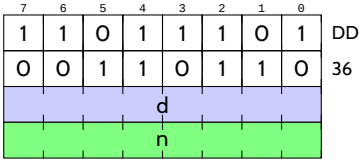
LD IX,(nn)



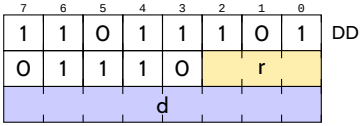
LD r,(IX+d)



LD (IX+d),n



LD (IX+d),r



LD IY,n

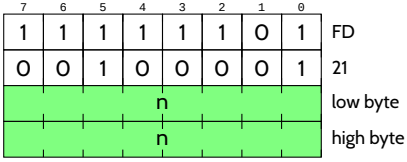
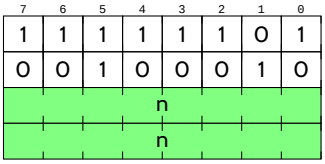
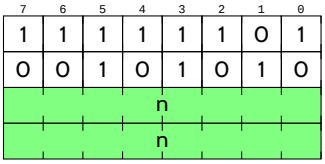


Table 22.44: Z80 Instruction Format LD (continued)

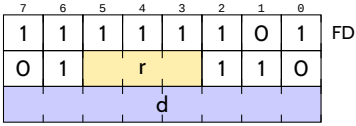
LD (nn),IY



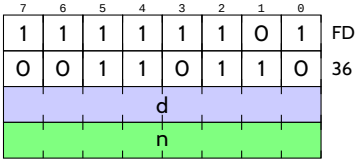
LD IY,(nn)



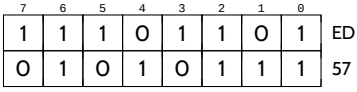
LD r,(IY+d)



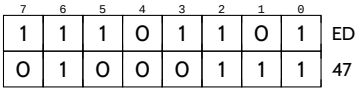
LD (IY+d),n



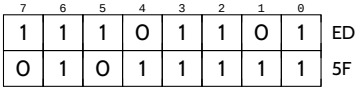
LD A,I



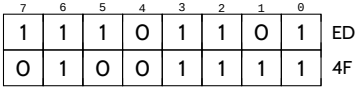
LD I,A



LD A,R



LD R,A



22.31 LDD
Block load with Decrement

Function $(DE) \leftarrow (HL)$

$DE \leftarrow DE - 1$

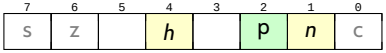
$HL \leftarrow HL - 1$

$BC \leftarrow BC - 1$

Description The contents of the memory location addressed by HL are loaded into the memory location addressed by DE.

Post op Decrement the DE, HL and BC registers after the copy.

Flags



h Cleared

n Cleared

p Reset if $BC = 0$ after execution, otherwise Set.

See also **LDDR**[42], **LDI**[43], **LDIR**[44]

42: **LDDR** Section 22.32 on page 332

43: **LDI** Section 22.33 on page 333

44: **LDIR** Section 22.34 on page 334

22.31.1 Block Load Instructions

Section 21.1 on page 271 covers this in more detail.

There are four instructions that load a block of memory which are listed below. The `LDI*` instructions increment the address so start from the beginning of the block whilst the `LDD*` instructions decrement the address so start from the end.

The `LD*R` instructions repeat the load until $BC = 0$.

The timings for those are for each iteration with the not taken timings for when a match or the end is reached.

The total timings will be dependent on the number of iterations taken.

Table 22.45: Z80 Instruction LDD

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
LDD	EDA8	2	4	16		

Table 22.46: Z80 Instruction Format LD__

	7	6	5	4	3	2	1	0	
	1	1	1	0	1	1	0	1	ED
LD_{F}	1	0	1	C	D	0	0	0	

Legend	
C	Repeat Instruction
0	run once
1	while $BC \neq 0$
D	Direction
0	Increment
1	Decrement

22.32 LDDR
Block load with Decrement

The Opcodes and timings are shown in Subsection 22.31.1

Function $(DE) \leftarrow (HL)$

$DE \leftarrow DE - 1$

$HL \leftarrow HL - 1$

$BC \leftarrow BC - 1$

Description The contents of the memory location addressed by HL are loaded into the memory location addressed by DE.

Post op Decrement the DE, HL and BC registers after the copy.

Repeat If $BC \neq 0$ then the program counter is decremented by two and the instruction is re-executed.

Flags

7	6	5	4	3	2	1	0
S	Z		*h*		*p*	*n*	C

h Cleared

n Cleared

p Cleared

See also **LDD**[45], **LDI**[46], **LDIR**[47]

The **LDIR** instruction is more commonly used than **LDDR** as it copies up from a start address rather than down.

45: **LDD** Section 22.31 on page 330

46: **LDI** Section 22.33 on page 333

47: **LDIR** Section 22.34 on page 334

22.33 LDI
Block load with Increment

The Opcodes and timings are shown in Subsection 22.31.1

Function $(DE) \leftarrow (HL)$

$DE \leftarrow DE + 1$

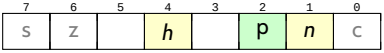
$HL \leftarrow HL + 1$

$BC \leftarrow BC - 1$

Description The contents of the memory location addressed by HL are loaded into the memory location addressed by DE.

Post op Increment the DE and HL registers, but decrement the BC register after the copy.

Flags



h Cleared

n Cleared

p Reset if $BC = 0$ after execution, otherwise Set.

See also **LDD**[48], **LDDR**[49], **LDIR**[50]

48: **LDD** Section 22.31 on page 330

49: **LDDR** Section 22.32 on page 332

50: **LDIR** Section 22.34 on page 334

22.34 LDIR
Block load with Increment

The Opcodes and timings are shown in Subsection 22.31.1

Function $(DE) \leftarrow (HL)$

$DE \leftarrow DE + 1$

$HL \leftarrow HL + 1$

$BC \leftarrow BC - 1$

Description The contents of the memory location addressed by HL are loaded into the memory location addressed by DE.

Post op Increment the DE and HL registers, but decrement the BC register after the copy.

Repeat If $BC \neq 0$ then the program counter is decremented by two and the instruction is re-executed.

Flags

7	6	5	4	3	2	1	0
S	Z		*h*		*p*	*n*	C

h Cleared

n Cleared

p Cleared

See also **LDD**[51], **LDDR**[52], **LDI**[53]

This is the most commonly used instruction in the Load block group of instructions. In games it's used to copy blocks of memory quickly - usually graphical items either directly into the screen or into a background frame prior to it being used for the screen.

51: **LDD** Section 22.31 on page 330

52: **LDDR** Section 22.32 on page 332

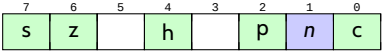
53: **LDI** Section 22.33 on page 333

22.35 NEG
Negate Accumulator

Function $A \leftarrow 0 - A$

Description The contents of the accumulator are subtracted from zero using two's complement and the result is stored back in the accumulator.

Flags



s Most significant bit of result is set

z Result was zero

h Carry from bit 3 to 4

n Set

p Set if A was `0x80`

c Set if A was `0x00` before the instruction

Table 22.47: Z80 Instruction NEG

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
NEG	ED44	2	2	8		

Table 22.48: Z80 Instruction Format NEG

	7	6	5	4	3	2	1	0	
NEG	1	1	1	0	1	1	0	1	ED
	0	1	0	0	0	1	0	0	44

22.36 NOP
No Operation

Function Delay

Description Nothing is done for 1 M cycle.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

This instruction is usually used to implement a timing delay of 1 M cycle for every **NOP** instruction executed. As this instruction takes 1 M cycle or 4 T states, then this instruction causes a delay of 2μs on a 2 MHz processor, or 4μs on a 1 MHz processor.

Table 22.49: Z80 Instruction Format NOP

	7	6	5	4	3	2	1	0	
NOP	0	0	0	0	0	0	0	0	00

22.37 OR
Bit or accomulator and operand

Function $A \leftarrow A \vee s$

Description The accumulator and the specified operand are logically or'ed and the result is stored in the accumulator.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	*c*

s Most significant bit of result is set

z Result was zero

p Parity of result is even

h Cleared

n Cleared

c Cleared

See also **AND**[54], **XOR**[55]

Figure 22.2: OR truth table

Memory \ Accumulator	0	1
0	0	1
1	1	1

54: **AND** Section 22.3 on page 280

55: **XOR** Section 22.67 on page 390

Table 22.50: Z80 Instruction OR

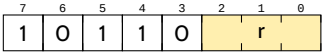
Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`OR a`	`B7`	1	1	4		
`OR b`	`B0`	1	1	4		
`OR c`	`B1`	1	1	4		
`OR d`	`B2`	1	1	4		
`OR e`	`B3`	1	1	4		
`OR h`	`B4`	1	1	4		
`OR l`	`B5`	1	1	4		
`OR N`	`F6`	2	2	7		
`OR (hl)`	`B6`	1	2	7		
`OR (ix+d)`	`DDB6nn`	3	5	19		
`OR (iy+d)`	`FDB6nn`	3	5	19		

Legend	
d	Data Offset
n	Immediate Data
r	Register

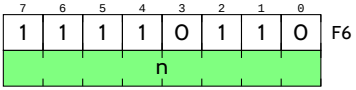
111	A
000	B
001	C
010	D
011	E
100	H
101	L

Table 22.51: Z80 Instruction Format OR

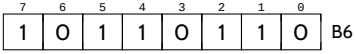
OR r



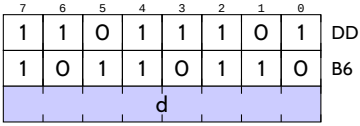
OR n



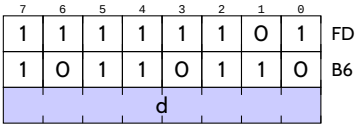
OR (HL)



OR (IX+d)



OR (IY+d)



22.38 OTDR Block output with Decrement

Section 21.4 on page 274 covers this in more detail.

The Opcodes and timings are shown in Subsection 22.41.1 on page 342

Function $(HL) \leftrightarrow (C)$

$B \leftarrow B - 1$

$HL \leftarrow HL - 1$

Description The peripheral device addressed by the C register is read and the result is loaded into the memory location addressed by the HL register pair.

Address bus The C register provides bits $A_0 \ldots A_7$ of the address bus.
B provides, after decrementation, bits $A_8 \ldots A_{15}$.

Post op The B register and the HL register pair are then each decremented.

Repeat If $B \neq 0$ then the program counter is decremented by 2 and the instruction is re-executed.

Flags

7	6	5	4	3	2	1	0
s	*z*		h		p	*n*	c

z Set

n Set

See also **OTIR**[56], **OUTD**[57], **OUTI**[58]

Note This instruction is named **OTDR** and not **OUTDR** as assemblers use a maximum of four characters for the instruction name.



Note that, unlike the block input instructions, the output block instructions decrement B before placing it onto the upper byte of the address bus.

56: OTIR Section 22.39 on page 340

57: OUTD Section 22.41 on page 343

58: OUTI Section 22.42 on page 345

22.39 OTIR
Block output with Increment

Section 21.4 on page 274 covers this in more detail.

The Opcodes and timings are shown in Subsection 22.41.1 on page 342

Function $(HL) \leftrightarrow (C)$

$B \leftarrow B - 1$

$HL \leftarrow HL + 1$

Description The peripheral device addressed by the C register is read and the result is loaded into the memory location addressed by the HL register pair.

Address bus The C register provides bits $A_0 \ldots A_7$ of the address bus.
B provides, after decrementation, bits $A_8 \ldots A_{15}$.

Post op The B register is decremented whilst the HL register pair is incremented.

Repeat If $B \neq 0$ then the program counter is decremented by 2 and the instruction is re-executed.

Flags

7	6	5	4	3	2	1	0
s	*z*		h		p	*n*	c

z Set

n Set

See also **OTDR**[59], **OUTD**[60], **OUTI**[61]

Note This instruction is named **OTIR** and not **OUTIR** as assemblers use a maximum of four characters for the instruction name.



Note that, unlike the block input instructions, the output block instructions decrement B before placing it onto the upper byte of the address bus.

59: OTDR Section 22.38 on page 339

60: OUTD Section 22.41 on page 343

61: OUTI Section 22.42 on page 345

22.40 OUT
Output register to Port

Name OUT (C),r

Description Output a register to the port C.

Address bus The C register provides bits $A_0 \ldots A_7$ of the address bus.
B provides bits $A_8 \ldots A_{15}$.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

Name OUT (N),A

Description Output the accumulator to the port N

Address bus The literal N provides bits $A_0 \ldots A_7$ of the address bus.
The Accumulator provides bits $A_8 \ldots A_{15}$.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

Table 22.52: Z80 Instruction OUT

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
OUT (c),a	ED79	2	3	12		
OUT (c),b	ED41	2	3	12		
OUT (c),c	ED49	2	3	12		
OUT (c),d	ED51	2	3	12		
OUT (c),e	ED59	2	3	12		
OUT (c),h	ED61	2	3	12		
OUT (c),l	ED69	2	3	12		
OUT (N),a	D3	1	3	11		

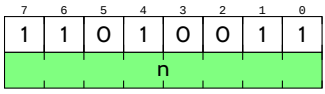
Legend

n	Immediate Data
r	Register

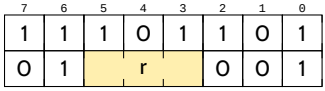
111	A
000	B
001	C
010	D
011	E
100	H
101	L

Table 22.53: Z80 Instruction Format OUT

OUT (N),A



OUT (C),r



22.41 OUTD
Block output with Decrement

Section 21.4 on page 274 covers this in more detail.

Function $(HL) \leftrightarrow (C)$

$B \leftarrow B - 1$

$HL \leftarrow HL - 1$

Description The peripheral device addressed by the C register is read and the result is loaded into the memory location addressed by the HL register pair.

Address bus The C register provides bits $A_0 \ldots A_7$ of the address bus.
B provides, after decrementation, bits $A_8 \ldots A_{15}$.

Post op The B register and the HL register pair are then each decremented.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	*n*	c

z Set if $B = 0$, reset if $B \neq 0$

n Set

See also **OTDR**[62], **OTIR**[63], **OUTI**[64]



Note that, unlike the block input instructions, the output block instructions decrement B before placing it onto the upper byte of the address bus.

62: **OTDR** Section 22.38 on page 339

63: **OTIR** Section 22.39 on page 340

64: **OUTI** Section 22.42 on page 345

Section 21.4 on page 274 covers this in more detail.

65: **OUTI** Section 22.42 on page 345

66: **OTIR** Section 22.39 on page 340

67: **OUTD** Section 22.41 on page 343

68: **OTDR** Section 22.38 on page 339

22.41.1 Output Block Instructions

There are four instructions which read from a block of memory and writes to a peripheral device. The **OUTI**[65] and **OTIR**[66] instructions increment the address so they start from the beginning of the block whilst the **OUTD**[67] and **OTDR**[68] instructions decrement the address so start from the end.

The `OT*R` instructions will automatically repeat until $B = 0$.

The timings for those are for each iteration with the not taken timings for when the end is reached.

The total timings will be dependent on the number of iterations taken.

Table 22.54: Z80 Instruction OUTD

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
OUTD	EDAB	2	4	16		

Table 22.55: Z80 Instruction Format OUT__

	7	6	5	4	3	2	1	0	
	1	1	1	0	1	1	0	1	ED
OUT␣	1	0	1	C	D	0	1	1	

Legend

C	Repeat Instruction
0	run once
1	while $BC \neq 0$
D	Direction
0	Increment
1	Decrement



The **OTDR** and **OTIR** instructions are so named instead of OUTDR or OUTIR as Z80 assemblers use a maximum of 4 characters for the instruction name

22.42 OUTI
Block output with Increment

Section 21.4 on page 274 covers this in more detail.

The Opcodes and timings are shown in Subsection 22.41.1 on page 342

Function $(HL) \leftrightarrow (C)$

$B \leftarrow B - 1$

$HL \leftarrow HL + 1$

Description The peripheral device addressed by the C register is read and the result is loaded into the memory location addressed by the HL register pair.

Address bus The C register provides bits $A_0 \ldots A_7$ of the address bus.
B provides, after decrementation, bits $A_8 \ldots A_{15}$.

Post op The B register is decremented whilst the HL register pair is incremented.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	*n*	c

z Set if $B = 0$, reset if $B \neq 0$

n Set

See also **OTDR**[69], **OTIR**[70], **OUTD**[71]



Note that, unlike the block input instructions, the output block instructions decrement B before placing it onto the upper byte of the address bus.

69: **OTDR** Section 22.38 on page 339

70: **OTIR** Section 22.39 on page 340

71: **OUTD** Section 22.41 on page 343

22.43 POP
Pop register pair from stack

Function $qq_{low} \leftarrow (SP)$

$qq_{high} \leftarrow (SP + 1)$

$SP \leftarrow SP + 2$

Description The contents of the memory location addressed by the stack pointer is loaded into the low order byte of the specified register pair and the stack register is incremented.

The contents of the memory location now addressed by the stack pointer is loaded into the high order byte of the register pair and the stack register is again incremented.

Flags

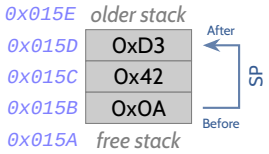
7	6	5	4	3	2	1	0
s	z		h		p	n	c

See also **PUSH**[72]

72: **PUSH** Section 22.44 on page 348

Figure 22.3 shows the stack before the instruction **POP** `BC` is executed.

Figure 22.3: Stack before and after pop



1. Initially the stack pointer is pointing to address `0x015B`.
2. **POP** then removes the first byte, placing it in the low order byte, in this instance the C register.
3. SP is then incremented and the next byte is placed in the high order byte, here the B register.
4. SP is incremented once more so it now points to `0x015D`.
5. The BC register pair now holds `0x420A`
   e.g. B `0x42` and C `0x0A`

In the case of `POP AF` the first byte removed is the flags and the second the accumulator.

Table 22.56: Z80 Instruction POP

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`POP af`	`F1`	1	3	10		
`POP bc`	`C1`	1	3	10		
`POP de`	`D1`	1	3	10		
`POP hl`	`E1`	1	3	10		
`POP ix`	`DDE1`	2	4	14		
`POP iy`	`FDE1`	2	4	14		

Table 22.57: Z80 Instruction Format POP

POP qq

7	6	5	4	3	2	1	0
1	1	q		0	0	0	1

POP IX

7	6	5	4	3	2	1	0	
1	1	0	1	1	1	0	1	DD
1	1	1	0	0	0	0	1	E1

POP IY

7	6	5	4	3	2	1	0	
1	1	1	1	1	1	0	1	FD
1	1	1	0	0	0	0	1	E1

Legend

q	Register Pair
`00`	BC
`01`	DE
`10`	HL
`11`	AF

22.44 PUSH
Push register pair onto stack

Function $(SP - 1) \leftarrow qq_{high}$

$(SP - 2) \leftarrow qq_{high}$

$SP \leftarrow SP - 2$

Description The stack pointer is decremented and the contents of the high order byte of the specified register pair is loaded into the memory location addressed by the stack pointer.

The stack pointer is again decremented and the low order byte of the register pair is loaded into the memory location now pointed to by the stack pointer.

Flags

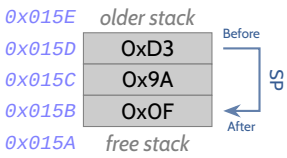
7	6	5	4	3	2	1	0
s	z		h		p	n	c

See also **PUSH**[73]

73: **PUSH** Section 22.44 on page 348

Figure 22.4 shows the stack after the instruction **PUSH** `DE` is executed.

Figure 22.4: Stack before and after push



1. We have the register pair DE set with the value `0x0A03`
   e.g. D `0x0A` and E `0x03`
2. Initially the stack pointer is pointing to address `0x015D`.
3. **PUSH** then decrements SP before loading the high order byte `0x0A` into address `0x015C`.
4. SP is then decremented again and the low order byte `0x03` is loaded into address `0x015B`.
5. SP now points to `0x015B` which is now the new top of the stack.

Table 22.58: Z80 Instruction PUSH

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`PUSH af`	`F5`	1	3	10		
`PUSH bc`	`C5`	1	3	10		
`PUSH de`	`D5`	1	3	10		
`PUSH hl`	`E5`	1	3	10		
`PUSH ix`	`DDE5`	2	4	15		
`PUSH iy`	`FDE5`	2	4	15		

Table 22.59: Z80 Instruction Format PUSH

PUSH qq

7	6	5	4	3	2	1	0
1	1	q		0	1	0	1

PUSH IX

7	6	5	4	3	2	1	0	
1	1	0	1	1	1	0	1	DD
1	1	1	0	0	1	0	1	E5

PUSH IY

7	6	5	4	3	2	1	0	
1	1	1	1	1	1	0	1	FD
1	1	1	0	0	1	0	1	E5

Legend

q	Register Pair
`00`	BC
`01`	DE
`10`	HL
`11`	AF

22.45 RES
Reset bit b in operand

Function $\overline{r_b} \leftarrow 0$

$\overline{(HL)_b} \leftarrow 0$

$\overline{(IX+d)_b} \leftarrow 0$

$\overline{(IY+d)_b} \leftarrow 0$

Description The specified bit of the operand is reset.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

See also BIT[74], SET[75]

74: **BIT** Section 22.4 on page 281

75: **SET** Section 22.62 on page 378

Table 22.60: Z80 Instruction RES

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
RES 0,a	CB87	2	2	8		
RES 0,b	CB80	2	2	8		
RES 0,c	CB81	2	2	8		
RES 0,d	CB82	2	2	8		
RES 0,e	CB83	2	2	8		
RES 0,h	CB84	2	2	8		
RES 0,l	CB85	2	2	8		
RES 0,(hl)	CB86	2	4	15		
RES 0,(ix+d)	DDCBnn86	4	6	23		
RES 0,(iy+d)	FDCBnn86	4	6	23		
RES 1,a	CB8F	2	2	8		
RES 1,b	CB88	2	2	8		
RES 1,c	CB89	2	2	8		
RES 1,d	CB8A	2	2	8		
RES 1,e	CB8B	2	2	8		
RES 1,h	CB8C	2	2	8		
RES 1,l	CB8D	2	2	8		
RES 1,(hl)	CB8E	2	4	15		
RES 1,(ix+d)	DDCBnn8E	4	6	23		

Table 22.60: Z80 Instruction RES (continued)

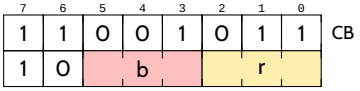
Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
RES 1,(iy+d)	FDCBnn8E	4	6	23		
RES 2,a	CB97	2	2	8		
RES 2,b	CB90	2	2	8		
RES 2,c	CB91	2	2	8		
RES 2,d	CB92	2	2	8		
RES 2,e	CB93	2	2	8		
RES 2,h	CB94	2	2	8		
RES 2,l	CB95	2	2	8		
RES 2,(hl)	CB96	2	4	15		
RES 2,(ix+d)	DDCBnn96	4	6	23		
RES 2,(iy+d)	FDCBnn96	4	6	23		
RES 3,a	CB9F	2	2	8		
RES 3,b	CB98	2	2	8		
RES 3,c	CB99	2	2	8		
RES 3,d	CB9A	2	2	8		
RES 3,e	CB9B	2	2	8		
RES 3,h	CB9C	2	2	8		
RES 3,l	CB9D	2	2	8		
RES 3,(hl)	CB9E	2	4	15		
RES 3,(ix+d)	DDCBnn9E	4	6	23		
RES 3,(iy+d)	FDCBnn9E	4	6	23		
RES 4,a	CBA7	2	2	8		
RES 4,b	CBA0	2	2	8		
RES 4,c	CBA1	2	2	8		
RES 4,d	CBA2	2	2	8		
RES 4,e	CBA3	2	2	8		
RES 4,h	CBA4	2	2	8		
RES 4,l	CBA5	2	2	8		
RES 4,(hl)	CBA6	2	4	15		
RES 4,(ix+d)	DDCBnnA6	4	6	23		
RES 4,(iy+d)	FDCBnnA6	4	6	23		
RES 5,a	CBAF	2	2	8		
RES 5,b	CBA8	2	2	8		
RES 5,c	CBA9	2	2	8		

Table 22.60: Z80 Instruction RES (continued)

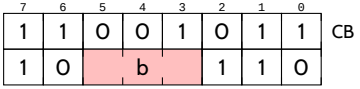
Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
RES 5,d	CBAA	2	2	8		
RES 5,e	CBAB	2	2	8		
RES 5,h	CBAC	2	2	8		
RES 5,l	CBAD	2	2	8		
RES 5,(hl)	CBAE	2	4	15		
RES 5,(ix+d)	DDCBnnAE	4	6	23		
RES 5,(iy+d)	FDCBnnAE	4	6	23		
RES 6,a	CBB7	2	2	8		
RES 6,b	CBB0	2	2	8		
RES 6,c	CBB1	2	2	8		
RES 6,d	CBB2	2	2	8		
RES 6,e	CBB3	2	2	8		
RES 6,h	CBB4	2	2	8		
RES 6,l	CBB5	2	2	8		
RES 6,(hl)	CBB6	2	4	15		
RES 6,(ix+d)	DDCBnnB6	4	6	23		
RES 6,(iy+d)	FDCBnnB6	4	6	23		
RES 7,a	CBBF	2	2	8		
RES 7,b	CBB8	2	2	8		
RES 7,c	CBB9	2	2	8		
RES 7,d	CBBA	2	2	8		
RES 7,e	CBBB	2	2	8		
RES 7,h	CBBC	2	2	8		
RES 7,l	CBBD	2	2	8		
RES 7,(hl)	CBBE	2	4	15		
RES 7,(ix+d)	DDCBnnBE	4	6	23		
RES 7,(iy+d)	FDCBnnBE	4	6	23		

Table 22.61: Z80 Instruction Format RES

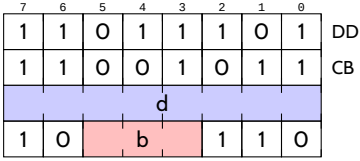
RES b,r



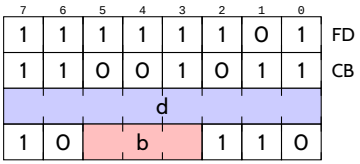
RES b,(HL)



RES b,(IX+d)



RES b,(IY+d)



Legend

b	Bit Number
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

d	Data Offset
r	Register
111	A
000	B
001	C
010	D
011	E
100	H
101	L

22.46 RET
Return from subroutine

Description Returns from a subroutine that was called by **CALL**.

The version taking a condition will only return from the subroutine if the condition is met.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

See also **CALL**[76], **RST**[77]

76: **CALL** Section 22.5 on page 283

77: **RST** Section 22.59 on page 371

Table 22.62: Z80 Instruction RET

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
RET	C9	1	3	10		
RET c	D8	1	3	11	1	5
RET m	F8	1	3	11	1	5
RET nc	D0	1	3	11	1	5
RET nz	C0	1	3	11	1	5
RET p	F0	1	3	11	1	5
RET pe	E8	1	3	11	1	5
RET po	E0	1	3	11	1	5
RET z	C8	1	3	11	1	5

Legend

c	Condition
000	NZ
001	Z
010	NC
011	C
100	PO
101	PE
110	P
111	M

Table 22.63: Z80 Instruction Format RET

RET

7	6	5	4	3	2	1	0
1	1	0	0	1	0	0	1

C9

RET c

7	6	5	4	3	2	1	0
1	1		c		0	0	0

22.47 RETI
Return from interrupt

Description Restore the contents of the Program Counter[78] analogous to the RET instruction.

Signal an I/O device that the interrupt routine has completed

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

See also **DI**[79], **EI**[80], **RETN**[81]

The **RETI** instruction also facilitates the nesting of interrupts, allowing higher priority devices to temporarily suspend service of lower priority service routines. However, this instruction does not enable interrupts that were disabled when the interrupt routine was entered.

Before doing the **RETI** instruction, the enable interrupt instruction **EI** should be executed to allow recognition of interrupts after completion of the current service routine.

79: **DI** Section 22.15 on page 298

80: **EI** Section 22.17 on page 301

81: **RETN** Section 22.48 on page 356

Table 22.64: Z80 Instruction RETI

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
RETI	ED4D	2	4	14		

Table 22.65: Z80 Instruction Format RETI

	7	6	5	4	3	2	1	0	
RETI	1	1	1	0	1	1	0	1	ED
	0	1	0	0	1	1	0	1	4D

22.48 RETN
Return from non-maskable interrupt

Description Restore the contents of the Program Counter[82].

Signal an I/O device that the interrupt routine has completed

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

See also **DI**[83], **EI**[84], **RETI**[85]

This instruction is used at the end of a non-maskable interrupts service routine to restore the contents of the Program Counter. The state of IFF2 is copied back to IFF1 so that maskable interrupts are enabled immediately following the **RETN** if they were enabled before the non-maskable interrupt.

Table 22.66: Z80 Instruction RETN

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
RETN	ED45	2	4	14		

Table 22.67: Z80 Instruction Format RETN

	7	6	5	4	3	2	1	0	
RETN	1	1	1	0	1	1	0	1	ED
	0	1	0	0	0	1	0	1	45

82: analogous to the RET instruction

83: **DI** Section 22.15 on page 298

84: **EI** Section 22.17 on page 301

85: **RETI** Section 22.47 on page 355

22.49 RL
Rotate left through carry

Function

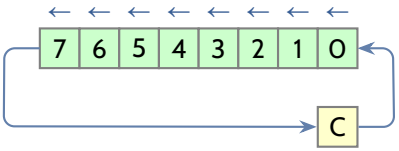


Figure 22.5: Rotate Left

Description The contents of the operand is rotated left 1 bit position through the Carry flag.

The previous contents of the Carry flag is copied to bit 0.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	c

s Most significant bit of result is set

z Result was zero

h Cleared

n Cleared

p Parity of result is even

c is the original bit 7 of the operand

See also **RLA**[86], **RLC**[87], **RLCA**[88], **SLA**[89]

The **RL A** instruction is similar to the 1-byte **RLA** except that here it also affects the s, z and p flags, whilst the other instruction does not. However, **RLA** is not just half the size but takes half the time to execute, so that instruction is better if speed is more important than knowing what the sign is or if it's zero.

86: **RLA** Section 22.50 on page 359

87: **RLC** Section 22.51 on page 360

88: **RLCA** Section 22.52 on page 362

89: **SLA** Section 22.63 on page 382

Table 22.68: Z80 Instruction RL

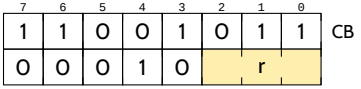
Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`RL a`	`CB17`	2	2	8		
`RL b`	`CB10`	2	2	8		
`RL c`	`CB11`	2	2	8		
`RL d`	`CB12`	2	2	8		
`RL e`	`CB13`	2	2	8		
`RL h`	`CB14`	2	2	8		
`RL l`	`CB15`	2	2	8		
`RL (hl)`	`CB16`	2	4	15		
`RL (ix+d)`	`DDCBnn16`	4	6	23		
`RL (iy+d)`	`FDCBnn16`	4	6	23		

Legend

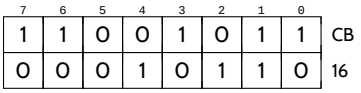
d	Data Offset
r	Register
`111`	A
`000`	B
`001`	C
`010`	D
`011`	E
`100`	H
`101`	L

Table 22.69: Z80 Instruction Format RL

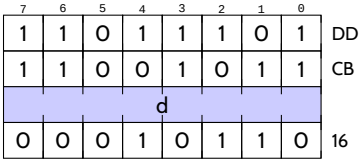
RL r



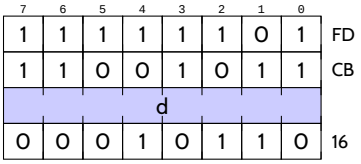
RL (HL)



RL (IX+d)



RL (IY+d)



22.50 RLA
Rotate A left through carry

Function

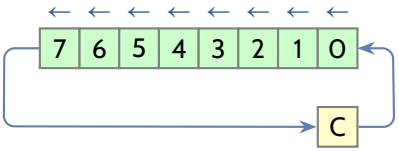


Figure 22.6: Rotate Left

Description The contents of the accumulator is rotated left 1 bit position through the Carry flag.

The previous contents of the Carry flag are copied to bit 0.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	c

h Cleared

n Cleared

c is the original bit 7 of the accumulator

See also **RL**[90]

90: **RL** Section 22.49 on page 357

Table 22.70: Z80 Instruction RLA

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
RLA	17	1	1	4		

Table 22.71: Z80 Instruction Format RLA

	7	6	5	4	3	2	1	0	
RLA	0	0	0	1	0	1	1	1	17

22.51 RLC
Rotate left with branch carry

Function

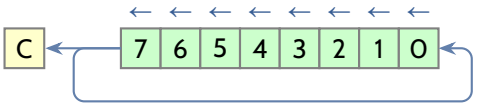


Figure 22.7: Rotate Left with branch carry

Description The contents of the operand is rotated left 1 bit position.

The original bit-7 is placed into both the Carry flag and bit-0.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	c

s Most significant bit of result is set

z Result was zero

h Cleared

n Cleared

p Parity of result is even

c is the original bit 7 of the operand

See also **RLCA**[91]

91: **RLCA** Section 22.52 on page 362

The **RLC A** instruction is similar to the 1-byte **RLCA** except that here it also affects the s, z and p flags, whilst the other instruction does not. However, **RLCA** is not just half the size but takes half the time to execute, so that instruction is better if speed is more important than knowing what the sign is or if it's zero.

Table 22.72: Z80 Instruction RLC

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`RLC a`	`CB07`	2	2	8		
`RLC b`	`CB00`	2	2	8		
`RLC c`	`CB01`	2	2	8		
`RLC d`	`CB02`	2	2	8		
`RLC e`	`CB03`	2	2	8		
`RLC h`	`CB04`	2	2	8		
`RLC l`	`CB05`	2	2	8		
`RLC (hl)`	`CB06`	2	4	15		
`RLC (ix+d)`	`DDCBnn06`	4	6	23		
`RLC (iy+d)`	`FDCBnn06`	4	6	23		

Table 22.73: Z80 Instruction Format RLC

RLC r

7	6	5	4	3	2	1	0	
1	1	0	0	1	0	1	1	CB
0	0	0	0	0	r			

RLC (HL)

7	6	5	4	3	2	1	0	
1	1	0	0	1	0	1	1	CB
0	0	0	0	0	1	1	0	06

RLC (IX+d)

7	6	5	4	3	2	1	0	
1	1	0	1	1	1	0	1	DD
1	1	0	0	1	0	1	1	CB
d								
0	0	0	0	0	1	1	0	06

RLC (IY+d)

7	6	5	4	3	2	1	0	
1	1	1	1	1	1	0	1	FD
1	1	0	0	1	0	1	1	CB
d								
0	0	0	0	0	1	1	0	06

Legend

d	Data Offset
r	Register

111	A
000	B
001	C
010	D
011	E
100	H
101	L

22.52 RLCA
Rotate A left with branch carry

Function

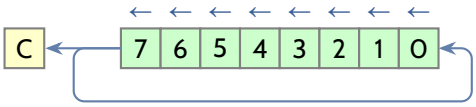


Figure 22.8: Rotate Left with branch carry

Description The contents of the accumulator is rotated left 1 bit position.

The original bit-7 is placed into both the Carry flag and bit-0.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	c

h Cleared

n Cleared

c is the original bit 7 of the accumulator

See also **RLC**[92]

92: **RLC** Section 22.51 on page 360

Table 22.74: Z80 Instruction RLCA

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
RLCA	07	1	1	4		

Table 22.75: Z80 Instruction Format RLCA

	7	6	5	4	3	2	1	0	
RLCA	0	0	0	0	0	1	1	1	07

22.53 RLD

Function

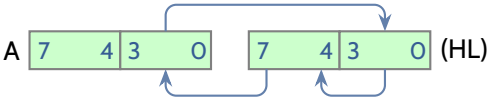


Figure 22.9: Rotate Left Decimal

The upper nibble of the accumulator is unchanged.

The Flags always refer to the Accumulator and not the memory location.

Description The low nibble of the memory addressed by HL is moved to the high nibble, whilst that moves to the low nibble of the accumulator. The low nibble of the accumulator moves to the low nibble of memory.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	c

s Most significant bit of result is set

z Result was zero

p Parity of result is even

h Cleared

n Cleared

See also **RRD**[93]

93: **RRD** Section 22.58 on page 370

Table 22.76: Z80 Instruction RLD

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
RLD	ED6F	2	5	18		

Table 22.77: Z80 Instruction Format RLD

	7	6	5	4	3	2	1	0	
RLD	1	1	1	0	1	1	0	1	ED
	0	1	1	0	1	1	1	1	6F

22.54 RR
Rotate right through carry

Function

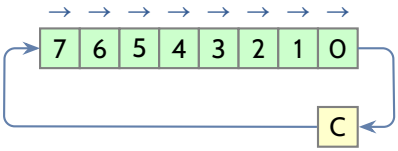


Figure 22.10: Rotate right

Description The contents of the operand is rotated right 1 bit position through the Carry flag.

The previous contents of the Carry flag is copied to bit 7.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	c

s Most significant bit of result is set

z Result was zero

p Parity of result is even

h Cleared

n Cleared

c is the original bit 0 of the operand

See also **RRA**[94]

94: **RRA** Section 22.55 on page 366

The **RR A** instruction is similar to the 1-byte **RRA** except that here it also affects the s, z and p flags, whilst the other instruction does not. However, **RRA** is not just half the size but takes half the time to execute, so that instruction is better if speed is more important than knowing what the sign is or if it's zero.

Table 22.78: Z80 Instruction RR

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
RR a	CB1F	2	2	8		
RR b	CB18	2	2	8		
RR c	CB19	2	2	8		
RR d	CB1A	2	2	8		
RR e	CB1B	2	2	8		
RR h	CB1C	2	2	8		
RR l	CB1D	2	2	8		
RR (hl)	CB1E	2	4	15		
RR (ix+d)	DDCBnn1E	4	6	23		
RR (iy+d)	FDCBnn1E	4	6	23		

Table 22.79: Z80 Instruction Format RR

RR r

7	6	5	4	3	2	1	0	
1	1	0	0	1	0	1	1	CB
0	0	0	1	1	r			

RR (HL)

7	6	5	4	3	2	1	0	
1	1	0	0	1	0	1	1	CB
0	0	0	1	1	1	1	0	1E

RR (IX+d)

7	6	5	4	3	2	1	0	
1	1	0	1	1	1	0	1	DD
1	1	0	0	1	0	1	1	CB
d								
0	0	0	1	1	1	1	0	1E

RR (IY+d)

7	6	5	4	3	2	1	0	
1	1	1	1	1	1	0	1	FD
1	1	0	0	1	0	1	1	CB
d								
0	0	0	1	1	1	1	0	1E

Legend

d	Data Offset
r	Register

111	A
000	B
001	C
010	D
011	E
100	H
101	L

22.55 RRA

Rotate A right through carry

Function

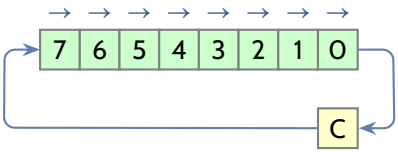


Figure 22.11: Rotate Left

Description The contents of the accumulator is rotated right 1 bit position through the Carry flag.

The previous contents of the Carry flag are copied to bit 0.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	c

c is the original bit 7 of the accumulator

h Cleared

n Cleared

See also **RR**[95]

95: **RR** Section 22.54 on page 364

Table 22.80: Z80 Instruction RRA

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
RRA	1F	1	1	4		

Table 22.81: Z80 Instruction Format RRA

	7	6	5	4	3	2	1	0	
RRA	0	0	0	1	1	1	1	1	1F

22.56 RRCA
Rotate A right with branch carry

Function

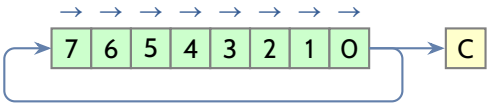


Figure 22.12: Rotate Right with branch carry

Description The contents of the accumulator is rotated right 1 bit position.

The original bit-0 is placed into both the Carry flag and bit-7.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	c

h Cleared

n Cleared

c is the original bit 0 of the accumulator

See also **RRC**[96]

96: **RRC** Section 22.57 on page 368

Table 22.82: Z80 Instruction RRCA

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
RRCA	0F	1	1	4		

Table 22.83: Z80 Instruction Format RRCA

	7	6	5	4	3	2	1	0	
RRCA	0	0	0	0	1	1	1	1	0F

22.57 RRC
Rotate right with branch carry

Function

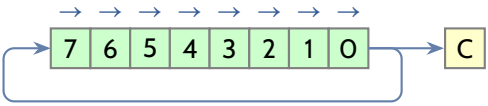


Figure 22.13: Rotate right with branch carry

Description The contents of the accumulator is rotated right 1 bit position.

The original bit-0 is placed into both the Carry flag and bit-7.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	c

s Most significant bit of result is set

z Result was zero

p Parity of result is even

h Cleared

n Cleared

c is the original bit 0 of the operand

See also **RRCA**[97]

97: **RRCA** Section 22.56 on page 367

The **RRC A** instruction is similar to the 1-byte **RRCA** except that here it also affects the s, z and p flags, whilst the other instruction does not. However, **RRCA** is not just half the size but takes half the time to execute, so that instruction is better if speed is more important than knowing what the sign is or if it's zero.

Table 22.84: Z80 Instruction RRC

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`RRC a`	`CB0F`	2	2	8		
`RRC b`	`CB08`	2	2	8		
`RRC c`	`CB09`	2	2	8		
`RRC d`	`CB0A`	2	2	8		
`RRC e`	`CB0B`	2	2	8		
`RRC h`	`CB0C`	2	2	8		
`RRC l`	`CB0D`	2	2	8		
`RRC (hl)`	`CB0E`	2	4	15		
`RRC (ix+d)`	`DDCBnn0E`	4	6	23		
`RRC (iy+d)`	`FDCBnn0E`	4	6	23		

Table 22.85: Z80 Instruction Format RRC

RRC r

7	6	5	4	3	2	1	0	
1	1	0	0	1	0	1	1	CB
0	0	0	0	1	r			

RRC (HL)

7	6	5	4	3	2	1	0	
1	1	0	0	1	0	1	1	CB
0	0	0	0	1	1	1	0	06

RRC (IX+d)

7	6	5	4	3	2	1	0	
1	1	0	1	1	1	0	1	DD
1	1	0	0	1	0	1	1	CB
d								
0	0	0	0	1	1	1	0	06

RRC (IY+d)

7	6	5	4	3	2	1	0	
1	1	1	1	1	1	0	1	FD
1	1	0	0	1	0	1	1	CB
d								
0	0	0	0	1	1	1	0	06

Legend

d	Data Offset
r	Register

111	A
000	B
001	C
010	D
011	E
100	H
101	L

22.58 RRD

Function

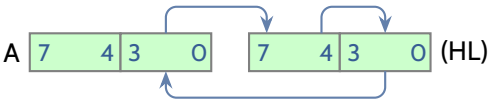


Figure 22.14: Rotate Right Decimal

The upper nibble of the accumulator is unchanged.

The Flags always refer to the Accumulator and not the memory location.

Description The low nibble of the memory addressed by HL is moved to the low nibble of the accumulator, whilst that moves to the high nibble of the memory. The original high nibble of the memory location moves to the low nibble of memory.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	c

s Most significant bit of result is set

z Result was zero

p Parity of result is even

h Cleared

n Cleared

See also **RLD**[98]

98: **RLD** Section 22.53 on page 363

Table 22.86: Z80 Instruction RRD

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
RRD	ED67					

Table 22.87: Z80 Instruction Format RRD

	7	6	5	4	3	2	1	0	
RRD	1	1	1	0	1	1	0	1	ED
	0	1	1	0	0	1	1	1	67

22.59 RST
Restart at p

Function $(SP - 1) \leftarrow PC_{high}$

$(SP - 2) \leftarrow PC_{low}$

$SP \leftarrow SP - 2$

$PC_{high} \leftarrow 0$

$PC_{low} \leftarrow p$

Description The contents of program counter is pushed onto the stack.

The value of p is placed into the program counter causing the processor to resume from that location.

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

See also **CALL**[99], **RET**[100]

The **RST** instructions provide eight single byte instructions which allow for a subroutine to be called very quickly.

Compared to **CALL** [101], **RST** runs 2 M cycles and 6 T states faster. Because of this, the **RST** instructions are used for either interrupt handlers or, for some operating systems, access to common routines like writing to the screen *etc.*.

To return from a **RST** call simply use the **RET** instruction, as the calling address is on the stack as if it was a **CALL** instruction making the call.

99: **CALL** Section 22.5 on page 283

100: **RET** Section 22.46 on page 354

101: **CALL** is 3-bytes, and takes 5 M cycles and 17 T states to run.

22.59.1 RST naming

The **RST** instructions had multiple different naming schemes which differ depending on who is writing the documentation.

Some wrote the instruction with the value of p as a decimal in the range 0 ... 7.

Others the address the RST instruction points to in decimal like 8, 16, 32 *etc.* The Sinclair documentation used this format.

Finally there were those who used hexadecimal like `0x8`, `0x10`, `0x18` *etc.*. This is the version I prefer as I rarely use decimal for memory addresses.

Zilog, however, are of no help here either as they use both decimal and hexadecimal schemes in their own documentation[2] about the Z80 instruction set.

[2] *Z80 CPU User Manual*, (Zilog Inc Aug. 2016) P.61,293

So, to save confusion, Table 22.88 shows all of these "conventions" in a single table along with an example use for each restart. The examples shown are for the Sinclair ZX Spectrum 48K ROM.

Table 22.88: RST Restarts

RST *p*			Call	Example use on the
p	dec	hex	Address	Sinclair Spectrum 48K
0	0	00	`0x0000`	Reset system
1	8	08	`0x0008`	Error handler
2	16	10	`0x0010`	Print character
3	24	18	`0x0018`	Get character
4	32	20	`0x0020`	Next character
5	40	28	`0x0028`	Calculator
6	48	30	`0x0030`	Allocate workspace
7	56	38	`0x0038`	Maskable Interrupt

22.59.2 RST `0x00`

This restart sets the program counter to `0x0000`.

This gets a special mention as this is the initial address the Z80 processor starts running from after a reset, so this instruction is commonly used as a means of a software reset.

On most systems, as the lower part of memory is represented by a ROM, then this vector usually resets the entire system.

However on some systems, the ROM is only present after a hardware reset and, as part of the boot process gets paged out with RAM.

An example of this is on the ZX Spectrum +3.

On this system, in normal use the ROM is present, but that model of Spectrum could run CP/M, so if CP/M was booted off a floppy disk, then the lower memory was paged out to RAM so that CP/M was present. Then the RST vectors were defined by that operating system and not the Spectrum's original OS.

The Spectrum 128K had multiple ROM's so the RST vectors had to cope with those.

The +2 and +3 machines could also page RAM at those locations.

22.59.3 RST `0x38`

This restart is also special in that, if the processor is in Interrupt Mode 1[102] then when a maskable interrupt occurs then this reset is invoked.

In the examples shown in Table 22.88 on the preceding page, the ZX Spectrum uses this interrupt mode and it's interrupt handler starts at address `0x0038`.

102: See **IM 1** Section 22.21 on page 306

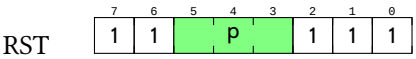
Table 22.89: Z80 Instruction RST

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
RST 0	C7	1	3	11		
RST 16	D7	1	3	11		
RST 24	DF	1	3	11		
RST 32	E7	1	3	11		
RST 40	EF	1	3	11		
RST 48	F7	1	3	11		
RST 56	FF	1	3	11		
RST 8	CF	1	3	11		

Legend

p	Restart
000	0x00 0
001	0x08 8
010	0x10 16
011	0x18 24
100	0x20 32
101	0x28 40
110	0x30 48
111	0x38 56

Table 22.90: Z80 Instruction Format RST



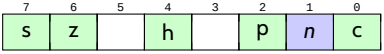
22.60 SBC
Subtract with borrow

Function $A \leftarrow A - s - C$

A,s

Description The specified operand and the carry is subtracted from the Accumulator with the result stored in the accumulator.

Flags



s Most significant bit of result is set

z Result was zero

h Carry from bit 3 to 4

p Parity of result is even

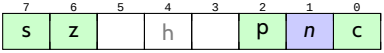
c Operation caused bit to be carried over

n Set

HL,ss

Description Subtract the register pair ss and the carry from HL, storing the result in HL.

Flags



s Most significant bit of result is set

z Result was zero

p Parity of result is even

c Operation caused bit to be carried over

n Set

See also **ADC**[103], **ADD**[104], **SUB**[105]

103: **ADC** Section 22.1 on page 276

104: **ADD** Section 22.2 on page 278

105: **SUB** Section 22.66 on page 388

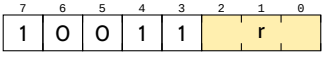
Legend	
d	Data Offset
n	Immediate Data
r	Register

111	A
000	B
001	C
010	D
011	E
100	H
101	L

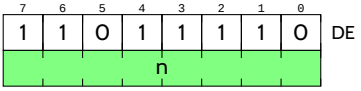
s	Register Pair
00	BC
01	DE
10	HL
11	SP

Table 22.91: Z80 Instruction Format SBC

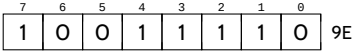
SBC r



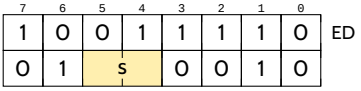
SBC n



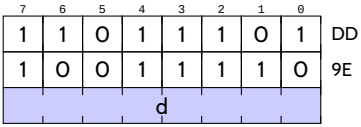
SBC (HL)



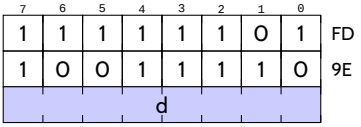
SBC HL,ss



SBC (IX+d)



SBC (IY+d)

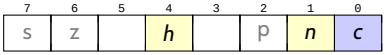


22.61 SCF
Set carry flag

Function $C \leftarrow 1$

Description Sets the carry flag.

Flags



c Set

h Cleared

n Cleared

See also **ADC**[106], **ADD**[107], **SUB**[108]

106: **ADC** Section 22.1 on page 276

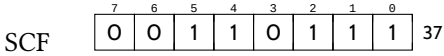
107: **ADD** Section 22.2 on page 278

108: **SUB** Section 22.66 on page 388

Table 22.92: Z80 Instruction SCF

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
SCF	37	1	1	4		

Table 22.93: Z80 Instruction Format SCF



22.62 SET
Set bit b in operand

Function $\overline{r_b} \leftarrow 1$

$\overline{(HL)_b} \leftarrow 1$

$\overline{(IX+d)_b} \leftarrow 1$

$\overline{(IY+d)_b} \leftarrow 1$

Description Sets the specified bit b in the operand s

Flags

7	6	5	4	3	2	1	0
s	z		h		p	n	c

See also **BIT**[109], **RES**[110]

109: **BIT** Section 22.4 on page 281

110: **RES** Section 22.45 on page 350

Table 22.94: Z80 Instruction SET

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
SET 0,a	CBC7	2	2	8		
SET 0,b	CBC0	2	2	8		
SET 0,c	CBC1	2	2	8		
SET 0,d	CBC2	2	2	8		
SET 0,e	CBC3	2	2	8		
SET 0,h	CBC4	2	2	8		
SET 0,l	CBC5	2	2	8		
SET 0,(hl)	CBC6	2	4	15		
SET 0,(ix+d)	DDCBnnC6	4	6	23		
SET 0,(iy+d)	FDCBnnC6	4	6	23		
SET 1,a	CBCF	2	2	8		
SET 1,b	CBC8	2	2	8		
SET 1,c	CBC9	2	2	8		
SET 1,d	CBCA	2	2	8		
SET 1,e	CBCB	2	2	8		
SET 1,h	CBCC	2	2	8		
SET 1,l	CBCD	2	2	8		
SET 1,(hl)	CBCE	2	4	15		
SET 1,(ix+d)	DDCBnnCE	4	6	23		

Table 22.94: Z80 Instruction SET (continued)

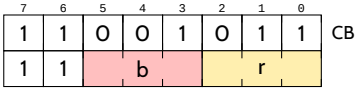
Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
SET 1,(iy+d)	FDCBnnCE	4	6	23		
SET 2,a	CBD7	2	2	8		
SET 2,b	CBD0	2	2	8		
SET 2,c	CBD1	2	2	8		
SET 2,d	CBD2	2	2	8		
SET 2,e	CBD3	2	2	8		
SET 2,h	CBD4	2	2	8		
SET 2,l	CBD5	2	2	8		
SET 2,(hl)	CBD6	2	4	15		
SET 2,(ix+d)	DDCBnnD6	4	6	23		
SET 2,(iy+d)	FDCBnnD6	4	6	23		
SET 3,a	CBDF	2	2	8		
SET 3,b	CBD8	2	2	8		
SET 3,c	CBD9	2	2	8		
SET 3,d	CBDA	2	2	8		
SET 3,e	CBDB	2	2	8		
SET 3,h	CBDC	2	2	8		
SET 3,l	CBDD	2	2	8		
SET 3,(hl)	CBDE	2	4	15		
SET 3,(ix+d)	DDCBnnDE	4	6	23		
SET 3,(iy+d)	FDCBnnDE	4	6	23		
SET 4,a	CBE7	2	2	8		
SET 4,b	CBE0	2	2	8		
SET 4,c	CBE1	2	2	8		
SET 4,d	CBE2	2	2	8		
SET 4,e	CBE3	2	2	8		
SET 4,h	CBE4	2	2	8		
SET 4,l	CBE5	2	2	8		
SET 4,(hl)	CBE6	2	4	15		
SET 4,(ix+d)	DDCBnnE6	4	6	23		
SET 4,(iy+d)	FDCBnnE6	4	6	23		
SET 5,a	CBEF	2	2	8		
SET 5,b	CBE8	2	2	8		
SET 5,c	CBE9	2	2	8		

Table 22.94: Z80 Instruction SET (continued)

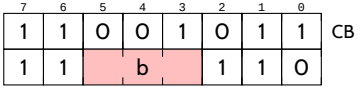
Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
SET 5,d	CBEA	2	2	8		
SET 5,e	CBEB	2	2	8		
SET 5,h	CBEC	2	2	8		
SET 5,l	CBED	2	2	8		
SET 5,(hl)	CBEE	2	4	15		
SET 5,(ix+d)	DDCBnnEE	4	6	23		
SET 5,(iy+d)	FDCBnnEE	4	6	23		
SET 6,a	CBF7	2	2	8		
SET 6,b	CBF0	2	2	8		
SET 6,c	CBF1	2	2	8		
SET 6,d	CBF2	2	2	8		
SET 6,e	CBF3	2	2	8		
SET 6,h	CBF4	2	2	8		
SET 6,l	CBF5	2	2	8		
SET 6,(hl)	CBF6	2	4	15		
SET 6,(ix+d)	DDCBnnF6	4	6	23		
SET 6,(iy+d)	FDCBnnF6	4	6	23		
SET 7,a	CBFF	2	2	8		
SET 7,b	CBF8	2	2	8		
SET 7,c	CBF9	2	2	8		
SET 7,d	CBFA	2	2	8		
SET 7,e	CBFB	2	2	8		
SET 7,h	CBFC	2	2	8		
SET 7,l	CBFD	2	2	8		
SET 7,(hl)	CBFE	2	4	15		
SET 7,(ix+d)	DDCBnnFE	4	6	23		
SET 7,(iy+d)	FDCBnnFE	4	6	23		

Table 22.95: Z80 Instruction Format SET

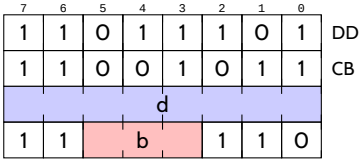
SET r



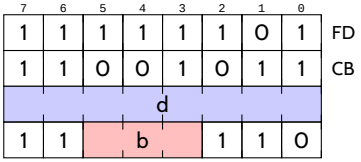
SET (HL)



SET (IX+d)



SET (IY+d)



Legend

b	Bit Number
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

d	Data Offset
r	Register
111	A
000	B
001	C
010	D
011	E
100	H
101	L

22.63 SLA
Arithmetic shift left

Function

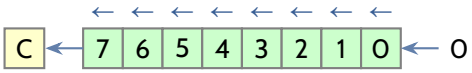


Figure 22.15: Arithmetic Shift Left

Description Shifts the operand left one bit with bit-7 being placed in the Carry flag.

A 0 is placed into bit-0.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	c

s Most significant bit of result is set

z Result was zero

p Parity of result is even

c Operation caused bit to be carried over

h Cleared

n Cleared

See also **RL**[111], **RLA**[112], **RLC**[113], **RLCA**[114]

111: **RL** Section 22.49 on page 357

112: **RLA** Section 22.50 on page 359

113: **RLC** Section 22.51 on page 360

114: **RLCA** Section 22.52 on page 362

Table 22.96: Z80 Instruction SLA

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`SLA a`	`CB27`	2	2	8		
`SLA b`	`CB20`	2	2	8		
`SLA c`	`CB21`	2	2	8		
`SLA d`	`CB22`	2	2	8		
`SLA e`	`CB23`	2	2	8		
`SLA h`	`CB24`	2	2	8		
`SLA l`	`CB25`	2	2	8		
`SLA (hl)`	`CB26`	2	4	15		
`SLA (ix+d)`	`DDCBnn26`	4	6	23		
`SLA (iy+d)`	`FDCBnn26`	4	6	23		

Table 22.97: Z80 Instruction Format SLA

SLA r

7	6	5	4	3	2	1	0	
1	1	0	0	1	0	1	1	CB
0	0	1	0	0	r			

SLA (HL)

7	6	5	4	3	2	1	0	
1	1	0	0	1	0	1	1	CB
0	0	1	0	0	1	1	0	26

SLA (IX+d)

7	6	5	4	3	2	1	0	
1	1	0	1	1	1	0	1	DD
1	1	0	0	1	0	1	1	CB
d								
0	0	1	0	0	1	1	0	26

SLA (IY+d)

7	6	5	4	3	2	1	0	
1	1	1	1	1	1	0	1	FD
1	1	0	0	1	0	1	1	CB
d								
0	0	1	0	0	1	1	0	26

Legend

d	Data Offset
r	Register

`111`	A
`000`	B
`001`	C
`010`	D
`011`	E
`100`	H
`101`	L

22.64 SRA
Arithmetic shift right

Function

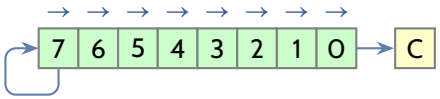


Figure 22.16: Arithmetic Shift Right

Description Shifts the operand right one bit with bit-0 being placed in the Carry flag.

Bit-7 is unchanged even though it is shifted into bit-6.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	c

s Most significant bit of result is set

z Result was zero

p Parity of result is even

c Operation caused bit to be carried over

h Cleared

n Cleared

See also **SRL**[115]

115: **SRL** Section 22.65 on page 386

This instruction is distinct amongst the shift right instructions in that it keeps bit-7 unchanged.

The reason for this is that when the operand is in two's complement and negative then the operand remains in the same sign even though it's value has been effectively divided by two.

To perform a 16-bit shift right, use **SRA** on the most significant byte, then **RR** on the least significant byte.

Table 22.98: Z80 Instruction SRA

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
SRA a	CB2F	2	2	8		
SRA b	CB28	2	2	8		
SRA c	CB29	2	2	8		
SRA d	CB2A	2	2	8		
SRA e	CB2B	2	2	8		
SRA h	CB2C	2	2	8		
SRA l	CB2D	2	2	8		
SRA (hl)	CB2E	2	4	15		
SRA (ix+d)	DDCBnn2E	4	6	23		
SRA (iy+d)	FDCBnn2E	4	6	23		

Table 22.99: Z80 Instruction Format SRA

SRA r

7	6	5	4	3	2	1	0	
1	1	0	0	1	0	1	1	CB
0	0	1	0	1	r			

SRA (HL)

7	6	5	4	3	2	1	0	
1	1	0	0	1	0	1	1	CB
0	0	1	0	1	1	1	0	2E

SRA (IX+d)

7	6	5	4	3	2	1	0	
1	1	0	1	1	1	0	1	DD
1	1	0	0	1	0	1	1	CB
d								
0	0	1	0	1	1	1	0	2E

SRA (IY+d)

7	6	5	4	3	2	1	0	
1	1	1	1	1	1	0	1	FD
1	1	0	0	1	0	1	1	CB
d								
0	0	1	0	1	1	1	0	2E

Legend

d	Data Offset
r	Register
111	A
000	B
001	C
010	D
011	E
100	H
101	L

22.65 SRL
Arithmetic shift right

Function

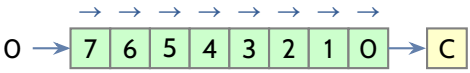


Figure 22.17: Arithmetic Shift Right

Description Shifts the operand right one bit with bit-0 being placed in the Carry flag.

A 0 is placed into bit-7.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	c

s Most significant bit of result is set

z Result was zero

p Parity of result is even

c Operation caused bit to be carried over

h Cleared

n Cleared

See also **SRA**[116]

116: **SRA** Section 22.64 on page 384

Table 22.100: Z80 Instruction SRL

Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
SRL a	CB3F	2	2	8		
SRL b	CB38	2	2	8		
SRL c	CB39	2	2	8		
SRL d	CB3A	2	2	8		
SRL e	CB3B	2	2	8		
SRL h	CB3C	2	2	8		
SRL l	CB3D	2	2	8		
SRL (hl)	CB3E	2	4	15		
SRL (ix+d)	DDCBnn3E	4	6	23		
SRL (iy+d)	FDCBnn3E	4	6	23		

Table 22.101: Z80 Instruction Format SRL

SRL r

7	6	5	4	3	2	1	0	
1	1	0	0	1	0	1	1	CB
0	0	1	1	1	r			

SRL (HL)

7	6	5	4	3	2	1	0	
1	1	0	0	1	0	1	1	CB
0	0	1	1	1	1	1	0	3E

SRL (IX+d)

7	6	5	4	3	2	1	0	
1	1	0	1	1	1	0	1	DD
1	1	0	0	1	0	1	1	CB
d								
0	0	1	1	1	1	1	0	3E

SRL (IY+d)

7	6	5	4	3	2	1	0	
1	1	1	1	1	1	0	1	FD
1	1	0	0	1	0	1	1	CB
d								
0	0	1	1	1	1	1	0	3E

Legend

d	Data Offset
r	Register

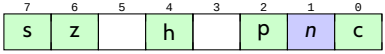
111	A
000	B
001	C
010	D
011	E
100	H
101	L

22.66 SUB
Subtract without carry

Function $A \leftarrow A - s$

Description The specified operand is subtracted from the Accumulator with the result stored in the accumulator.

Flags



s Most significant bit of result is set

z Result was zero

h Carry from bit 3 to 4

p Parity of result is even

c Operation caused bit to be carried over

n Set

See also **ADC**[117], **ADD**[118], **SBC**[119]

117: **ADC** Section 22.1 on page 276

118: **ADD** Section 22.2 on page 278

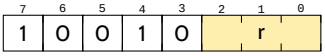
119: **SBC** Section 22.60 on page 375

Table 22.102: Z80 Instruction SUB

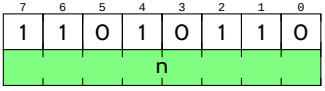
Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
SUB a,a	97	1	1	4		
SUB a,b	90	1	1	4		
SUB a,c	91	1	1	4		
SUB a,d	92	1	1	4		
SUB a,e	93	1	1	4		
SUB a,h	94	1	1	4		
SUB a,l	95	1	1	4		
SUB a,(hl)	96	1	2	7		
SUB N	D6	2	2	7		
SUB (ix+d)	DD96nn	3	5	19		
SUB (iy+d)	FD96nn	3	5	19		

Table 22.103: Z80 Instruction Format SUB

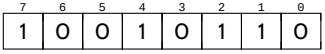
SUB r



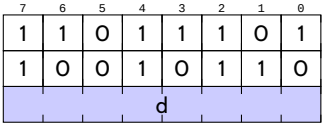
SUB n



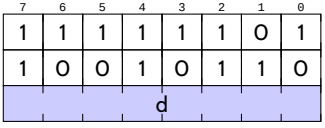
SUB (HL)



SUB (IX+d)



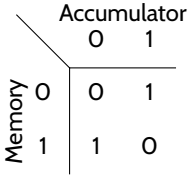
SUB A,(IY+d)



Legend	
d	Data Offset
n	Immediate Data
r	Register
111	A
000	B
001	C
010	D
011	E
100	H
101	L

22.67 XOR
Exclusive or

Figure 22.18: Exclusive OR truth table



Function $A \leftarrow A\ s$

Description The accumulator and the specified operand are exclusive or'ed and the result is stored in the accumulator.

Flags

7	6	5	4	3	2	1	0
s	z		*h*		p	*n*	c

s Most significant bit of result is set

z Result was zero

p Parity of result is even

h Cleared

n Cleared

c Cleared

See also **AND**[120], **OR**[121]

120: **AND** Section 22.3 on page 280

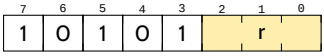
121: **OR** Section 22.37 on page 337

Table 22.104: Z80 Instruction XOR

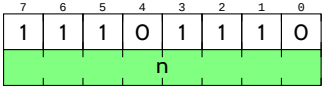
Opcode	Hex	Length	M cycles	T states	M not taken	T not taken
`XOR a`	AF	1	1	4		
`XOR b`	A8	1	1	4		
`XOR c`	A9	1	1	4		
`XOR d`	AA	1	1	4		
`XOR e`	AB	1	1	4		
`XOR h`	AC	1	1	4		
`XOR l`	AD	1	1	4		
`XOR N`	EE	2	2	7		
`XOR (hl)`	AE	1	2	7		
`XOR (ix+d)`	DDAEnn	3	5	19		
`XOR (iy+d)`	FDAEnn	3	5	19		

Table 22.105: Z80 Instruction Format XOR

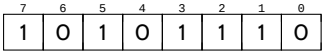
XOR r



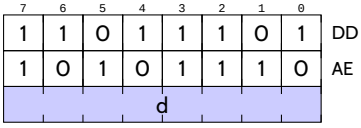
XOR n



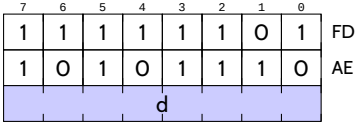
XOR (HL)



XOR (IX+d)



XOR A,(IY+d)



Legend	
d	Data Offset
n	Immediate Data
r	Register
`111`	A
`000`	B
`001`	C
`010`	D
`011`	E
`100`	H
`101`	L

Page intentionally blank

Assemblers

Table of contents

23 BBC Basic Assembler **397**
23.1 Address modes . . . 398
23.2 Assembly Destination . . . 401
23.3 Multiple Passes . . . 403
23.4 Directives . . . 405

24 BeebASM **409**
24.1 Installing BeebASM . . . 410
24.2 Command line . . . 411
24.3 Source file syntax . . . 412
24.4 Variables . . . 416
24.5 Operators . . . 417
24.6 Directives . . . 421
24.7 Contants . . . 438
24.8 Functions . . . 439

25 Lancaster Assembler **447**
25.1 Using the 65C02 assembler Version 1.60 . . . 447
25.2 Introduction . . . 448
25.3 The Assembler's CLI mode . . . 449
25.4 Changing screen mode . . . 449
25.5 Creating source files . . . 450
25.6 Starting an assembly . . . 452
25.7 Source Program Format . . . 461
25.8 The Output Listing . . . 469
25.9 Directives Defining Data and Constants . . . 479
25.10 Object File Control Directives . . . 487
25.11 Source File Control Directives . . . 493
25.12 Conditional Assembly . . . 495
25.13 Progress Reporting . . . 499
25.14 Writing Simple Macros . . . 503
25.15 The Macro Programming Language . . . 509
25.16 Supported Processors . . . 526
25.17 Directives . . . 529
25.18 Differences from the ADE Assembler . . . 554

25.19 Differences from previous releases 555
25.20 Customising the assembler 557

Page intentionally blank

23 BBC Basic Assembler

BBC BASIC comes with its own 6502 assembler[1] built in.

Assembly language statements within a **BASIC** program must be enclosed between a pair of [ ]. When the **BASIC** program is run, the assembly language statements between the square brackets are assembled into machine code.

The address where the machine code is inserted into the memory location is usually specified by the *P%*. Section 23.2 on page 401 covers this in full detail.

Each assembly language statement consists of an optional label (which must start with a dot), a 3-letter instruction mnemonic and the address mode for the specific instruction. If a label is specified, it must be separated from the instruction by at least one space. The mnemonic and the address mode does not necessarily require a space.

Comments can be included by use of a backslash (\) character. Everything after that character until either the next colon or end of line will be ignored by the assembler.

Listing 23.1 on the following page shows an example of a simple assembly language program.

1: BBC Basic on a Z80 machine comes with a Z80 assembler.
e.g. **BBC BASIC** on the Z80 coprocessor or on a clone machine like the Wren Professional

Listing 23.1: Example 6502 assembly language program in BBC Basic

```
 10 OSBYTE=&FFF4
 20 P%=&3500
 30 [
 40 .start LDA #129 \load accumulator with
    OSBYTE number
 50 LDX #100
 60 LDY #0
 70 JSR OSBYTE
 80 RTS
 90 ]
100 CALL &3500
```

23.1 Address modes

The **BBC BASIC** assembler supports the standard address modes of the 6502 microprocessor.

23.1.1 Implicit

Implicit addressing is where a constant value between 0...255 is passed to a register. The constant is prefixed with a #.

Line 40 of listing Listing 23.1 on the next page loads the accumulator with the value 129, whilst line 50 sets X with 100 and 60 sets Y with 0.

23.1.2 Absolute address

A absolute address is a 16-bit address of a memory location anywhere within the 64K address range of the 6502 processor. Unlike Implicit[2], there is no prefix to the address.

2: Implicit Subsection 23.1.1 on page 398

23.1.3 zero-page address

Similar to Absolute[3] addressing, the zero-page (addresses `0x0000` ...`0x00FF`) have their own address mode which accepts the address as a single byte. This is specified in the same manner as Absolute addressing, except that the assembler will, if the instruction supports it, use this address mode which is one byte shorter.

3: Absolute
Subsection 23.1.2 on page 398

23.1.4 Indirect

Indirect addressing is where an address is specified by the instruction but it is the contents of that location which holds the required information.

For example,

```
JMP (&2010)
```

means to jump to the address stored at location `0x2010`.

23.1.5 Indexed

Indexed addressing takes an address and with that address plus one of the X or Y index registers, get the actual memory location for the instruction.

```
LDA &1F00,X
```

will set the accumulator to the value in the address $0x1F00 + X$

23.1.6 zero-page Indexed

zero-paged indexed addressing is similar, just that the final address will be within the zero-page.

zero-page Indexed X

```
LDA (&82,X)
```

$addr = 0x82 + X$
$A = (addr)$

takes the address 0x82, adds X to it and then uses the contents of that address as the address containing the value to set the accumulator to.

zero-page Indexed Y

```
LDA (&80),Y
```

$addr = (0x82) + Y$
$A = (addr)$

takes the address stored in location 0x80, then add Y to that to get the value to set the accumulator to.

23.1.7 Relative

The branch instructions use relative addressing which can point to the next instruction within 127 bytes forward or 128 bytes behind the branch instruction.

23.1.8 Accumulator

The final addressing mode supported by the **BBC BASIC** assembler is Accumulator, where the Accumulator is actually referenced by the instruction.

```
ASL A
```

is the ASL instruction, but it shifts left the accumulator rather than a memory location.

23.2 Assembly Destination

The **BBC BASIC** assembler will store the assembly at the location of the *P%* variable unless the *O%* variable is also set.

When *O%* is set and the value of OPT[4] is 5...7 the assembler will assemble the code to run from the location set by *P%* but store the code at *O%*.

4: OPT Table 23.1 on page 407

This allows for programs to be assembled to run at an alternate location which is either currently occupied or is not writeable.

For example:

- Run in the location the **BBC BASIC** program would normally be stored, *e.g.* Games from Disk or Tape from 0x0E00
- Run as a Paged Rom located from 0x8000 upwards.

Listing Listing 23.2 on the next page shows a simple example assembling and storing at a specific location.

20 Create 100 bytes in which to store our machine code program
30 Create 0x1C bytes for storing the data for the machine code
40 Runs a FOR loop for the integer variable *opt%*[5] twice, with the values 0 then 3
50 Sets the destination as the GAP array.
160 is the end of the FOR loop
170 The data the loop will send to OSWRCH, stored in data
240 Executes the assembly routine from the label entry

5: *opt%* Figure 23.1 on page 406

Listing 23.2: Example 6502 assembly language program in BBC Basic

```
 10 OSWRCH=&FFF4
 20 DIM GAP% 100
 30 DIM data &1C
 40 FOR opt%=0 TO 3 STEP 3
 50 P%=GAP%
 60 [
 70 OPT opt%
 80 .entry LDX#0
 90 .loop LDA data,X
100 JSR OSWRCH
110 INX
120 CPX #&1C
130 BNE loop
140 RTS
150 ]
160 NEXT opt%
170 !data=&01190416
180 data!4=&00C800C8
190 data!8=&00C80119
200 data!&C=&01190000
210 data!&10=&00ADFF9C
220 data!&14=&FF9C0119
230 data!&18=&0000FF53
240 CALL entry
```

23.3 Multiple Passes

Like most assemblers[6], the **BBC BASIC** assembler does not support forward references unless you run the assembler across the source multiple times.

For example, Listing 23.3 on page 403 shows a program which will fail with `No such variable at line 70` because the label zero is referenced before it is declared:

6: Some assemblers hide this behind the scenes

```
 10 DIM GAP 50
 20 address=&70
 30 P%=GAP
 40 [
 50 LDA address
 60 CMP #0
 70 BEQ zero
 80 STA address+1
 90 .zero RTS
100 ]
```

Listing 23.3: Broken assembly language program with forward reference

Line 7: This will fail because zero is forward of this point

The solution to this simple problem that all assemblers[7] use is to run the assembler across the source twice.

7: Also other languages have this issue, even TEX, run multiple passes to resolve forward references.

The first pass usually runs without reporting any errors, allowing for labels to get defined when they are declared, then a second pass which will then fail if anything is missing.

BBC BASIC supports this with the OPT statement. Figure Figure 23.1 on page 406 shows how each value of OPT affects the assembler.

So to fix Listing 23.3 we run the assembler twice. Listing 23.4 shows the mchanges

Listing 23.4: Working assembly language program with forward reference

Line 3: Run pass with values 0 & 3

Line 6: Set OPT to set the pass

Line 9: This now works as zero declared in first pass

Line 13: Loop to second pass

```
10 DIM GAP 50
20 address=&70
30 FOR pass=0 TO 3 STEP 3
35 P%=GAP
40 [
45 OPT pass
50 LDA address
60 CMP #0
70 BEQ zero
80 STA address+1
90 .zero RTS
100 ]
110 NEXT pass
```

Here we enclose the assembly language inside a loop which performs two passes, one where pass=0, the second with pass=3. Remember Figure Figure 23.1 on page 406 shows the value values for OPT.

23.4 Directives

23.4.1 [ ]

Syntax [*source code lines*]

Description Mark the start and end of an assembly language program

The characters [ ] delimitate the assembly language program from the rest of the **BBC BASIC** program containing it.

23.4.2 EQUB

Syntax **EQUB** value

EQUB variable

Description Equate Byte stores a one-byte, 8-bit value into the program.

EQUB is a pseudo-operation which allows data to be incorporated into the body of an assembly language program.

If the value passed to it, take the least significant portion if the value exceeds the size of the **EQU** directive. For example, if the value passed to **EQUB** is greater than 255 then the lower 8-bits will be used.

23.4.3 EQUW

Syntax **EQUW** value

EQUW variable

Description Equate Word stores a two-byte, 16-bit value into the program.

Similar to **EQUB**[8], if the value exceeds 2 bytes, the lower significant portion is used.

8: **EQUB** Subsection 23.4.2 on page 405

23.4.4 EQUD

Syntax **EQUD** value

EQUD variable

Description Equate Double Word stores a four-byte, 32-bit value into the program.

Similar to **EQUB**[9], if the value exceeds 4 bytes, the lower significant portion is used.

9: **EQUB** Subsection 23.4.2 on page 405

23.4.5 EQUS

Syntax **EQUS** string

EQUS variable$

Description Equate String stores a string into the program.

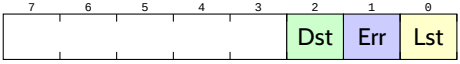
The value passed to **EQUS** can be either a "quoted"string or a string variable like A$.

23.4.6 OPT

Syntax **OPT** option

Description **OPT** sets the options for the assembler.

Figure 23.1 shows the values of the **OPT** statement within the **BBC BASIC** assembler whilst Table 23.1 on the next page shows its effect on the assembler.



Dst Destination, 0=P%, 1=O%
Err Errors, 1=report errors, 0=suppress
Lst Listing, 1=show listing, 0=no listing

Figure 23.1: BBC Basic Assembler OPT values

OPT	Destination	Errors	Listing
0	*P%*	suppressed	no listing
1			listing
2		reported	no listing
3			listing
4	*O%*	suppressed	no listing
5			listing
6		reported	no listing
7			listing

Table 23.1: BBC Basic Assembler OPT actions

P% the destination address where the assembled code will execute.

O% where the assembled code will be stored for values of OPT of 4 ... 7.
Otherwise, the code will be stored at P% where it will be executed.

Page intentionally blank

24 BeebASM

BeebASM is a portable 6502 assembler with BBC Micro style syntax which runs under Linux, MacOS and Windows.

Its syntax is reminiscent of **BBC BASIC**'s built-in assembler[1] and can write it's output directly to DFS disk images ready for use in an emulator or on real hardware via a GoTek device.

Although BeebASM was written for the BBC Micro I have used it for building software for the Commodore C64. In theory, it should also work for any other 6502 or 65C02 based system.

1: assembler Chapter 23 on page 397

24.1 Installing BeebASM

Before you can compile and install BeebASM you need to ensure you have a working build environment:

This applies for Debian or Ubuntu based distributions which use the apt package managers.

Other package managers should be similar.

```
1 sudo apt install build-essential g++ cmake git
```

24.1.1 Compilation

If you got this book as part of the bundled zip or tar archive then you can use the sources in the following to build beebasm:

Line 1: Directory within the zip/tar file containing BeebASM sources

```
1 cd
2 mkdir build
3 cd build
4 cmake ..
5 make
```

Alternatively you can use the following to get the current version from GitHub:

Listing 24.1: Cloning BeebAsm from GitHub

```
1 git clone
2 cd beebasm
3 mkdir build
4 cd build
5 cmake ..
6 make
```

24.1.2 Installation

Once compiled with either of the above methods you can then install it into your system with:

```
1 sudo make install
```

24.2 Command line

Name beebasm

Format **beebasm** [*options*] **-i** *filename*

Options

-i *file*	Specify source filename
-o *file*	Specify output filename (when not specified by SAVE command)
-di *file*	Specify a disc image file to be added to
-do *file*	Specify a disc image file to output
-boot *file*	Specify a filename to be run by !BOOT on a new disc image
-opt *opt*	Specify the *OPT 4,n for the generated disc image
-title *title*	Specify the title for the generated disc image
-v	Verbose output
-d	Dump all global symbols after assembly
-w	Require whitespace between opcodes and labels
-vc	Use Visual C++-style error messages
-D *sym*=*val*	Define symbol prior to assembly

24.3 Source file syntax

Listing 24.2 on page 412 shows a simple example of a source file.

Listing 24.2: Source Example

```
    CPU 0           ; 6502
    ORG &2000       ; Start address
; Used to mark start of file for SAVE on line 18
.start

; A traditional assembly comment
.initialise
    LDY #31
    LDA #0          ; Value to store in buffer
.loop               \ Loop
    STA buffer,Y
    DEY:BPL loop
    RTS

; Save everything generated between lines 4 and
    17
; into a file.
.end
    SAVE "TEST", start, end
```

Line 7: Label of subroutine entry point

Line 10: Alternate comment syntax

Line 12: Multiple instructions on a single line

Line 18: Save the output to a file

24.3.1 Comments

Comments are introduced by a semicolon or backslash. Unlike the BBC Micro assembler, these continue to the end of the line, and are not terminated by a colon.

24.3.2 Instructions

Assembler instructions are written with the standard 6502 syntax.

Instructions can be written one-per-line, or many on one line, separated by colons.

24.3.3 Labels

A label is defined by preceding it with a ".", as per the BBC Micro assembler, *e.g.*.loop A label need not be followed by a colon.

24.3.4 Local labels

Curly braces can be used to specify a block of code in which all symbols and labels defined will exist only within this block. Effectively, this is a mechanism for providing 'local labels' without the slightly cumbersome syntax demanded by some other assemblers.

These can be nested, as visible in Listing 24.5 on page 415.

Any symbols defined within a block will override any of the same name outside the block, exactly like C/C++.

```
1  .initialise
2  {
3      LDY #31
4      LDA #0
5  .loop
6      STA buffer,Y
7      DEY:BPL loop
8      RTS
9  }
10
11 .copy
12 {
13     LDY #31
14 .loop
15     LDA source,Y
16     STA dest,Y
17     DEY:BPL loop
18     RTS
19 }
```

Listing 24.3: Local labels

Line 5: label visible only within the braces

Line 14: perfectly ok to define *loop* again in a new block

Labels can be defined within a scope which exist outside that scope in two ways depending on how they are prefixed.

Globally visible local labels

The prefix '`.*`' indicates the label be globally visible.

Listing 24.4 on the next page shows an example of this where the label `loop` is only visible within the subroutine but the label `copy_x` has been marked as global so can be referenced from anywhere.

Listing 24.4: Global label within local scope

Line 4: copy_x will be visible globally

Line 6: loop is only visible within the braces

Line 17: *copy_x* is visible here so this works

```
1  .copy_10
2  {
3      LDX #10
4  .*copy_x
5      DEX
6  .loop
7      LDA from,X
8      STA to,X
9      DEX:BPL loop
10     RTS
11 }
12
13     ; Call  copy_10 to make 10 copies
14     JSR copy_10
15
16     ; Call copy_x to make 20 copies
17     LDX #20:JSR copy_x
```

Define label visible in the parent scope

The prefix '.^' indicates that the label is declared in the parent scope:

Listing 24.5 on the facing page shows a subrouting which has two levels of localized labels.

The inner block between lines 4...11 has a single label defined called `loop`.

That label is only visible between lines 2...16 but not globally.

```
1  .contrived
2  {
3      LDX #255
4      {
5          LDY #3
6  .^loop
7          DEX
8          BEQ exit
9          DEY
10         BNE loop
11     }
12     LDY #7
13     JMP loop
14 .exit
15     RTS
16 }
17     JMP loop
```

Listing 24.5: Define label in parent scope

Line 13: loop *is* visible here

Line 17: error generated as loop is *not* visible here

24.4 Variables

Variables are defined using the BASIC syntax:

```
1 addr = &70
2 name = "Bob"
3 surname = "Quoted""string"
```

Line 3: example of " inside a string

Variables are immutable. Once a variable had been declared it cannot be redefined. Attempting to redeclare a varable will cause an error.

The only exception to this is in **FOR**...**NEXT** statement blocks as they use a different variable scope.

24.4.1 Conditional assignment

Variables can be declared if they have not already been defined using the **=?** syntax.

This is useful when using the **-D** command line option to allow for defaults to be set in the source but allowing them to be overridden on the command line.

```
1 addr =? &70
```

24.5 Operators

Table 24.1 on page 417 lists the available operators.

Table 24.1: Numeric operators

Op	Action
+	Addition
-	Subtraction
*	Division
/	Multiplication
^	Raise to the power of.
DIV	Integer division.
MOD	Integer modulus.
«	Arithmetic shift left
»	Arithmetic shift right
AND	Bitwise AND.
OR	Bitwise OR.
EOR	Bitwise EOR.
= or ==	Equals
<> or !=	Not Equal
<	Less than
>	Greater Than
<=	Less Than or Equal
>=	Greater Than or Equal
( ) or [ ]	Parentheses
-	Negate (unary minus)
< and >	16 to 8-bit conversions

- Except for () [] and negate, all of the operators use the form a **op** b
- The exception's are: (a), [a], -a, <a and >a
- The comparison operators return 0 for false or -1 for true.
- [a] is implemented as an alternate to (a) to avoid confusion with 6502 indirect instructions.

24.5.1 16-bit to 8-bit conversions

BeebASM supports two operations to convert a 16-bit expression into two 8-bit values to make it easier to include within instructions.

< val = val MOD 256

<*val* or LO(*val*)

Return the least-significant byte of a 16-bit expression, specifically bits 0...7.

> val = val DIV 256

>val or HI(val)

Return the most-significant byte of a 16-bit expression, specifically bits 8...15.

Listing 24.6: LO/HI example

```
.demo
    LDX #<message       ; X is low 8-bits
    LDY #>message       ; Y is high 8-bits
    JSR printMessage    ; Print the message
    RTS

.message
    EQUS "Hello world", 0

; Print 0 terminated string at address XY
.printMessage
{
    STX &70     ; Store address in zero page
    STY &71
    LDY #0      ; Start from byte 0
.loop
    LDA (&70),Y ; Get next character
    BEQ exit    ; Exit if A=0
    JSR &FFEE   ; Print character to screen
    INY         ; Increment Y and loop until we
    hit
    BNE loop    ; 256 to prevent an infinite
    loop
.exit
```

```
23      RTS
24  }
```

24.5.2 Numeric Literals

Literal
0
1
1.2E10
1e-7
&70
$70
&FFEE
%01100110

Numeric literals are in decimal by default, and can be integers or reals.

Scientific notation can be used, *e.g.* 1.2E10 or 1e-7.

Hex literals are prefixed with "&" or "$".

Binary literals are prefixed with "%".

Digits can be separated with an underscore to improve readability, *e.g.* 1_000_000 or $1_0000.

A character in single quotes (*e.g.*'A') returns its ASCII code.

24.5.3 String Literals

Strings are delimited with double quotes.

To include Quotes inside a string then they need to be doubled.

Listing 24.7: Literals

Line 2: Numeric literals

Line 6: String literal

Line 9: Quote inside string

Line 12: null terminated string

```
1  .examp1
2      EQUB    1, 2, 3, 4
3      EQUD    1e4, 1E5
4
5  .helloWorld
6      EQUS    "Hello World!"
7
8  .quoteExample
9      EQUS    "This is ""quoted""!"
10
11 .terminatedString
12     EQUS    "Welcome", 0
```

24.6 Directives

24.6.1 ALIGN

Description Align the current assembly address to a boundary

Syntax `ALIGN alignment`

Example `ALIGN &100`

Used to align the address pointer to the next boundary, *e.g.* use **ALIGN** &100 to move to the next page which is useful for positioning a table at a page boundary so that index accesses don't incur a "page crossed" penalty.

24.6.2 ASM

Description Assemble the supplied assembly language string.

Syntax `ASM str`

Example `ASM "LDA #&41"`

24.6.3 ASSERT

Description Abort assembly if any of the expressions is false.

Syntax `ASSERT a [, b, c, …]`

See also **ERROR**[2]

2: **ERROR**
Subsection 24.6.11 on page 424

24.6.4 CLEAR

Description Clears all guards between the *start* and *end* addresses specified.

Syntax `CLEAR start, end`

See also GUARD[3]

3: GUARD Subsection 24.6.13 on page 426

This can also be used to reset a section of memory which has had code assembled in it previously.

BeebAsm will complain if you attempt to assemble code over previously assembled code at the same address without having CLEARed it first.

24.6.5 COPYBLOCK

Description Copies a block of assembled data from one location to another

Syntax `COPYBLOCK start, end, dest`

This is useful to copy code assembled at one location into a program's data area for relocation at run-time.

24.6.6 CPU

Description Selects the target CPU

Syntax `CPU n`

Example `CPU 1`

See also CPU[4]

4: CPU Subsection 24.7.2 on page 438

Table 24.2: CPU options

Option	CPU
0	6502
1	65C02

24.6.7 EQUB

Description Insert the specified bytes into the code.

Syntax `EQUB a [, b, c, ...]`

See also EQUD[5], EQUS[6], EQUW[7]

Unlike BBC BASIC, that a comma-separated sequence including strings can be inserted.

Under the surface, there is no difference between EQUB and EQUS, both can take bytes or string parameters.

5: **EQUD** Subsection 24.6.8 on page 423

6: **EQUS** Subsection 24.6.9 on page 423

7: **EQUW** Subsection 24.6.10 on page 423

24.6.8 EQUD

Description Insert the specified 32-bit word(s) into the code.

Syntax `EQUD a [, b, c, ...]`

See also EQUB[8], EQUS[9], EQUW[10]

8: **EQUB** Subsection 24.6.7 on page 423

9: **EQUS** Subsection 24.6.9 on page 423

10: **EQUW** Subsection 24.6.10 on page 423

24.6.9 EQUS

Description Inserts the specified string into the code

Syntax `EQUS "a " [, "b ", c, ...]`

See also EQUB[11], EQUD[12], EQUW[13], MAPCHAR[14]

As **EQUS** can take byte parameters, to zero-terminate a string, you can write:

```
1 EQUS "My string", 0
```

11: **EQUB** Subsection 24.6.7 on page 423

12: **EQUD** Subsection 24.6.8 on page 423

13: **EQUW** Subsection 24.6.10 on page 423

14: **MAPCHAR** Subsection 24.6.18 on page 432

24.6.10 EQUW

Description Insert the specified 16-bit wordyte(s) into the code.

Syntax `EQUW a [, b, c, ...]`

See also EQUB[15], EQUD[16], EQUS[17]

15: **EQUB** Subsection 24.6.7 on page 423

16: **EQUD** Subsection 24.6.8 on page 423

17: **EQUS** Subsection 24.6.9 on page 423

24.6.11 ERROR

Description Abort assembly with the provided error message.

Syntax `ERROR "message "`

See also ERROR[18]

18: ERROR
Subsection 24.6.3 on page 421

This can be useful for enforcing certain constraints on generated code, for example:

```
.table
FOR n, 1, 32
    EQUB 255 / n
NEXT

IF HI(P%) <> HI(table)
    ERROR "Table crosses page boundary"
ENDIF
```

24.6.12 FOR

Description Perform a loop over a section of code

Syntax `FOR var, start, end [, step ]`

Syntax `; assembly directives…`

Syntax `NEXT`

FOR works exactly like BASIC's FOR…NEXT in that it repeats the directives within the loop a number of times.

For example, in listing Listing 24.8, the variable n only exists for the scope of the FOR…NEXT loop. Also, any labels or variables defined within the loop are only visible within it.

Listing 24.8: FOR NEXT example

```
; loop with n = 0, 2, 4, 6, 8, 10
FOR n, 0, 10, 2
    PRINT n
    LDA #0:STA &900+n
    LDA #n:STA &901+n
```

```
6 | NEXT
```

However, unlike BBC BASIC, forward references to labels inside the loop will work properly, so, for example, this little multiply routine is perfectly ok:

Listing 24.9: FOR NEXT with forward references

```
\\ multiplies A*X, puts result in product/
    product+1
.multiply
    CPX #0:BEQ zero
    DEX:STX product+1
    LSR A:STA product:LDA #0
    FOR n, 0, 7
        BCC skip
        ADC product+1
.skip
        ROR A:ROR product
    NEXT
    STA product+1:RTS
.zero
    STX product:STX product+1:RTS
```

Line 9: This would break BBC BASIC

24.6.13 GUARD

Description Puts a 'guard' on the specified address which will cause an error if you attempt to assemble code over this address.

Syntax `GUARD addr`

Example Listing Listing 24.10 on page 427 shows an example of **GUARD** in action.

See also **CLEAR**[19]

19: **CLEAR** Subsection 24.6.4 on page 422

24.6.14 IF

Description Perform assembly based on various conditions

Syntax `IF condition`

Syntax `; assembly directives…`

Syntax `ELIF condition`

Syntax `; assembly directives…`

Syntax `ELSE`

Syntax `; assembly directives…`

Syntax `ENDIF`

Example Listing Listing 24.10 on the facing page

These statements can be placed on one line, separated by colons. However, unlike BBC Basic, **ENDIF** must be present to denote the end of the **IF** block.

```
1  \\ build a rather strange table
2  FOR n, 0, 9
3      IF (n AND 1) = 0
4          a = n*n
5      ELSE
6          a = -n*n
7      ENDIF
8      EQUB a
9  NEXT
10
11 // IF in a single line
12 IF debugraster:LDA #3:STA &FE21:ENDIF
13
14 // Determine destination based on compile type
15 IF rom
16     ORG &8000
17     GUARD &C000
18 ELIF tube
19     ORG &B800
20     GUARD &F800
21 ELSE
```

Listing 24.10: IF ENDIF examples

Line 17: Prevents code assembling outside the usable area of a ROM. See **??** on page ??

```
22      ORG &3C00
23      GUARD &7C00
24  ENDIF
```

24.6.15 INCBIN

Description Includes the specified binary file in the object code at this point.

Syntax `INCBIN "filename "`

24.6.16 INCLUDE

Description Includes the specified source file in the code at this point.

Syntax `INCLUDE "filename "`

24.6.17 MACRO

Description Define assembler macro

Syntax `MACRO name [,parameter list… ]`

Syntax `; Macro definition`

Syntax `ENDMACRO`

MACRO is best illustrated by an example. This defines a macro called ADDI8 (”ADD Immediate 8-bit”) whose function is to add a constant to a memory address. It expects two parameters: the memory address and the constant value.

Listing 24.11: Macro example

```
1  MACRO ADDI8 addr, val
2      IF val=1
3          INC addr
4      ELIF val>1
5          CLC
6          LDA addr
7          ADC #val
8          STA addr
9      ENDIF
10 ENDMACRO
```

The body of the macro contains an IF block which will generate the most appropriate code according to the constant value passed in.

Then, at any point afterwards, the macro can be used as follows:

```
ADDI8 &900, 1           ; add 1 to address &900
ADDI8 bonus, 10         ; add 10 to the bonus
ADDI8 pills, pill_add   ; pills += pill_add
```

Listing Listing 24.12 on page 430 shows how macros can be called from other macros.

Listing 24.12: Calling Macro's from Macro's

```
MACRO ADDI16 addr, val
    IF val=0
        ; do nothing
    ELIF val=1
        INC addr
        BNE skip1
        INC addr+1
.skip1
    ELIF HI(val)=0
        ADDI8 addr, val
        BCC skip2
        INC addr+1
.skip2
    ELSE
        CLC
        LDA addr
        ADC #LO(val)
        STA addr
        LDA addr+1
        ADC #HI(val)
        STA addr+1
    ENDIF
ENDMACRO
```

Care should be taken with macros, as the details of the code assembled are hidden.

In the above ADDI16 example, the C flag is not set consistently, depending on the inputs to the macro.

e.g. it remains unchanged if val=0 or 1, and will not be correct if val<256.

24.6.18 MAPCHAR

Description Change mapping of characters from ASCII

Syntax `MAPCHAR ascii code, remapped code`

Syntax `MAPCHAR start code, end code, remapped code`

See also EQUB[20], EQUS[21]

20: EQUB Subsection 24.6.7 on page 423

21: EQUS Subsection 24.6.9 on page 423

By default, when EQUS "string " is assembled, the ASCII codes of each character are written into the object code. MAPCHAR allows you to specify which value should be written to the object code for each character.

Suppose you have a font which contains the following symbols - space, followed by A-Z, followed by digits 0-9, followed by .,!?-'

You could specify this with MAPCHAR as follows:

Listing 24.13: MAPCHAR example

```
1 MAPCHAR ' ', 0
2 MAPCHAR 'A','Z', 1
3 MAPCHAR '0','9', 27
4 MAPCHAR '.', 37
5 MAPCHAR ',', 38
6 MAPCHAR '!', 39
7 MAPCHAR '?', 40
8 MAPCHAR '-', 41
9 MAPCHAR ''', 42
```

Now, when writing strings with EQUS, these codes will be written out instead of the default ASCII codes.

24.6.19 ORG

Description Set the address to be assembled from.

Syntax `ORG addr`

Example `ORG &8000`

This can be changed multiple times during a source file if you wish (for example) to assemble two separate blocks of code at different addresses, but share the labels between both blocks.

This is exactly equivalent to BBC BASIC's P%=<addr>.

24.6.20 PRINT

Description Displays some text

Syntax `PRINT arg ...`

PRINT takes a comma-separated list of strings or values.

To print a value in hex, prefix the expression with a ' ' character.

```
1 PRINT "Value of label 'start' =", ~start
2 PRINT "numdots =", numdots, "dottable size =",
      dotend-dotstart
```

Listing 24.14: Print Examples

You can also use the following within **PRINT** commands:

FILELINE$ show the current file and line number.

CALLSTACK$** same as **FILENAME$ but also show all parent macro and include files as well.

24.6.21 PUTBASIC

Description Tokenize a BBC Basic program and save it on the output disc image

Syntax
```
PUTBASIC host filename, [beeb filename
]
```

See also PUTFILE[22], PUTTEXT[23], SAVE[24]

This takes a BBC BASIC program as a plain text file on the host OS, tokenises it, and outputs it to the disc image as a native BASIC file.

Line numbers can be provided in the text file if desired, but if not present they will be automatically generated.

22: PUTFILE
Subsection 24.6.22 on page 434
23: PUTTEXT
Subsection 24.6.23 on page 435
24: SAVE
Subsection 24.6.25 on page 436

24.6.22 PUTFILE

Description Copy a file from the host to the output disc image

Syntax
```
PUTFILE host filename,
[beeb filename,]
start addr [,exec addr ]
```

See also PUTBASIC[25], PUTTEXT[26], SAVE[27]

If no *beeb filename* is provided, the host filename will be used and must therefore be 7 characters or less in length.

A start address must be provided, and optionally an execution address can be provided.

25: PUTBASIC
Subsection 24.6.21 on page 434
26: PUTTEXT
Subsection 24.6.23 on page 435
27: SAVE
Subsection 24.6.25 on page 436

24.6.23 PUTTEXT

Description Copy a text file from the host to the output disc image

Syntax
```
PUTTEXT host filename,
[beeb filename,]
start addr [,exec addr ]
```

See also PUTBASIC[28], PUTFILE[29], SAVE[30]

This command is the same as **PUTFILE**, except that the host file is assumed to be a text file and its line endings will be automatically converted to CR (the BBC standard line ending) from any of CR, LF, CRLF or LFCR.

28: PUTBASIC
Subsection 24.6.21 on page 434

29: PUTFILE
Subsection 24.6.22 on page 434

30: SAVE
Subsection 24.6.25 on page 436

24.6.24 RANDOMIZE

Description

Syntax
```
RANDOMIZE n
```

See also RND[31]

Seed the random number generator used by the **RND** function.

If this is not used, the random number generator is seeded based on the current time and so each build of a program using **RND** will be different.

31: RND ??

24.6.25 SAVE

Description Saves out object code to either a DFS disc image (if one has been specified), or to the current directory as a standalone file.

Syntax
```
SAVE ["filename ",] start, end
[, exec [, reload ] ]
```

See also PUTBASIC[32], PUTFILE[33], PUTTEXT[34]

A source file must have at least one **SAVE** statement in it, otherwise nothing will be output. BeebAsm will warn if this is the case.

filename is optional only if a name is specified with **-o** on the command line.

exec can be optionally specified as the execution address of the file when saved to a disc image.

reload can additionally be specified to save the file on the disc image to a different address to that which it was saved from. Use this to assemble code at its 'native' address, but which loads at a DFS-friendly address, ready to be relocated to its correct address upon execution.

32: PUTBASIC
Subsection 24.6.21 on page 434
33: PUTFILE
Subsection 24.6.22 on page 434
34: PUTTEXT
Subsection 24.6.23 on page 435

24.6.26 SKIP

Description Moves the address pointer on by the specified number of bytes. Use this to reserve a space of a fixed size in the code.

Syntax
```
SKIP bytes
```

24.6.27 SKIPTO

Description Moves the address pointer to the specified address. An error is generated if this address is behind the current address pointer.

Syntax
```
SKIPTO addr
```

24.6.28 SOURCELINE

Description For error reporting act as if the following lines started at line number in filename.

Syntax
```
SOURCELINE
line number [,filename ]
```

24.7 Contants

A few constants are also defined by beebasm.

24.7.1 * or P%

Description Returns the current address within the assembly

24.7.2 CPU

Description Returns the value from the CPU directive.

Example `0 or 1`

See also CPU[35]

35: CPU Subsection 24.6.6 on page 422

24.7.3 PI

Description returns the value of **PI**

Example `3.1415927…`

24.7.4 FALSE

Description returns 0

Example `0`

24.7.5 TRUE

Description returns -1

Example `-1`

24.8 Functions

Functions whose name ends with $ return a string otherwise a numeric result.

24.8.1 ABS

Description the absolute value of val

Syntax `ABS (val)`

24.8.2 ACS

Description the arc-cosine of val

Syntax `ACS (val)`

24.8.3 ASC

Description the numeric ASCII value of the first character in str

Syntax `ASC (str)`

Example `ASC ("A") = 65`

24.8.4 ASN

Description the arc-sine of val

Syntax `ASN (val)`

24.8.5 ATN

Description the arc-tangent of val

Syntax `ATN (val)`

24.8.6 CHR$

Description the ASCII character whose value is val

Syntax `CHR$ (val)`

Example `CHR$ (64) = "@"`

24.8.7 COS

Description the cosine of val

Syntax `COS (val)`

24.8.8 DEG

Description convert radians to degrees

Syntax `DEG (val)`

Function $val \times \frac{180}{\pi}$

24.8.9 EXP

Description e raised to the power of val

Syntax `EXP (val)`

Function e^{val}

24.8.10 EVAL

Description evaluates the expression in str

Syntax `EVAL (str)`

24.8.11 INT

Description round val to the integer nearest zero

Syntax `INT (val)`

24.8.12 LEFT$

Description returns the first length characters of str

Syntax `LEFT$ (str, length)`

Example `LEFT$ ("ABCDE", 3) = "ABC"`

24.8.13 LEN

Description the length of str

Syntax `LEN (str)`

Example `LEN ("ABC") = 3`

24.8.14 LOWER$

Description returns str in lowercase

Syntax `LOWER$ (str)`

Example `LOWER$ ("ABCD") = "abcd"`

24.8.15 LN

Description the natural, base e, logarithm of val

Syntax `LN (val)`

Function $\log_e(val)$

24.8.16 LOG

Description the base 10 logarithm of val

Syntax `LOG (val)`

Function $\log_{10}(val)$

24.8.17 MID$

Description returns the specified substring of str

Syntax `MID$ (str, index, length)`

Example `MID$ ("ABCDE", 2, 3) = "BCD"`

index is one based so the first character is 1 not 0.

24.8.18 NOT

Description the Ones-complement[36] of val

Syntax `NOT (val)`

36: Ones-complement
Subsection 7.8.2 on page 73

24.8.19 RAD

Description convert degrees to radians

Syntax `RAD (val)`

Function $val \times \frac{\pi}{180}$

24.8.20 RIGHT$

Description return the last length characters of str

Syntax `RIGHT$ (str, length)`

Example `RIGHT$ ("ABCDE, 3) = "CDE"`

24.8.21 RND

Description generate a random number

Syntax `RND (val)`

See also RANDOMIZE[37]

RND (1) will return a random number between 0 ... 1.

RND (n) returns an integer between 0 ... $n - 1$.

37: RANDOMIZE
Subsection 24.6.24 on page 435

24.8.22 SIN

Description the sine of val

Syntax `SIN (val)`

24.8.23 SGN

Description return -1, 0 or 1 depending on if *val* is negative, zero or positive

Syntax `SGN (val)`

Example `SGN (-10) = -1`

Example `SGN (0) = 0`

Example `SGN (3.1415) = 1`

24.8.24 SQR

Description square root of val.

Syntax `RAD (val)`

Function $\sqrt{val}$

24.8.25 STR$

Description converts val into a numeric string

Syntax `STR$ (val)`

Example `STR$ (42) = "42"`

See also STR$~

24.8.26 STR$~

Description converts val into a hexadecimal string.

Syntax `STR$~ (val)`

Example `STR$~ (42) = "2A"`

See also STR$

24.8.27 STRING$

Description returns the count copies of str

Syntax `STRING$ (count, str)`

Example `STRING$ (3, "AB") = "ABABAB"`

24.8.28 TIME$

TIME$ will use the same value of date/time throughout the entire assembly

Description Return the current time in the specified format

Syntax `TIME$ (format)`

Example `TIME$ ("HH:MM:SS")`

Example `TIME$ ()`

Example `TIME$`

format must be one supported by the C library strftime() function.

When used as **TIME$** without parentheses or a format then the format "Day,DD Mon Year.HH:MM:SS" is used.

24.8.29 TAN

Description Tangent of val.

Syntax `TAN (val)`

24.8.30 UPPER$

Description returns str in uppercase

Syntax `UPPER$ (str)`

Example `UPPER$ ("abcd) = "ABCD"`

24.8.31 VAL

Description returns the decimal value of str

Syntax `VAL (str)`

Example `VAL ("123) = 123`

Page intentionally blank

25 Lancaster Assembler

The Lancaster Assembler is a Paged ROM which supports both the original 6502 and the 65C02 processors.Although it was targetted for the BBC Model B and the Master 128, it did work on the Acorn Electron as I did use it throughout the late 1980's on an Electron where I had sideways ram available.

This section covers version 1.60 of this ROM and is mostly taken from the original documentation. All I have done is taken the ASCII text and converted it to LaTeX.

25.1 Using the 65C02 assembler Version 1.60

The 65C02 Assembler is copyright (C) Alan Phillips 1987.

It may be passed on by anyone, to anyone, and used for any peaceful purpose whether private or commercial.

No purchase, licensing or permission is needed.

It may be distributed in any way or posted on any Bulletin Board system provided that it is not sold (apart from reasonable handling and media costs), that due credit is made for authorship, and that this paragraph is brought to the attention of the recipient.

Edition 3.1
October 1987
Alan Phillips

Acknowledgments

Thanks are due to Dave Morriss, Neil Mercer, Alan Baker, Peter Vince, Mike Tubby, Scott Larson and Peter Francis for many helpful suggestions and comments during the development of the Assembler.

25.2 Introduction

This guide describes how to use the 65C02 Assembler produced by Alan Phillips of the Department of Computing at Lancaster University. The reader is assumed to be familiar with the concepts of programming at assembly level.

The 65C02 Assembler runs in sideways RAM or as a sideways ROM on the BBC Models B, B+, B+128, Master 128 and Master Compact, with or without a 6502 Second Processor. It supports all the opcodes of the 6502 and 65C02 processor families, and contains a powerful Macro Programming Language.

Remember I have tested this on an Electron so it works there as well

The Assembler is a disc-oriented system. Source files must be heldon disc, and the object code will be written to disc. It can be used with the Acorn DFS, ADFS, NFS and any other Acorn-compatible disc or network filing system, but not on cassette-only machines.

There are no source editing facilities contained in the Assembler. However, it is able to accept source produced by most text editors or word-processor programs.

25.2.1 Installing the Assembler

The 65C02 Assembler code is completely unprotected. If you have a machine with sideways RAM, you will be able to load it to a RAM bank and run it; if not, you will need to program it into a 27128 EPROM and fit it into a sideways ROM socket.

25.3 The Assembler's CLI mode

If you have the Assembler fitted into the highest priority ROM[1] slot of your machine, pressing **BREAK** or powering on will enter it as the current language. In this case the Assembler is said to be in "CLI (Command Language Interpreter) mode".

1: On the *Master series* *CONFIG LANG can set any language ROM as the initial language on power on.
§ Section 28.13 on page 579

You will see a "*" prompting you for input, and any lines you type will be sent to the BBC Operating System as MOS commands. In this mode there is no need to type a "*" yourself in front of the commands, although it won't matter if you do.

You can also enter CLI mode by issuing the command *CLI regardless of the ROM socket the Assembler is in.

From CLI mode you can of course start an assembly with the *ASSEMBLE [2] command. Alternatively, you could enter another language by typing, say, *BASIC, *VIEW or *WORDWISE

2: *ASSEMBLE
Section 25.6 on page 452

Naturally, you are able to issue DFS commands in CLI mode, or you could *TYPE files, and so on.

25.4 Changing screen mode

You can't use the **BASIC** statement MODE to change screen mode, though, since the current language is not **BASIC**. However, for convenience the Assembler will respond to a command *MODE, so you could type

```
*MODE 3
```

to set mode 3 if you wished.

This command is available either from the Assembler's CLI mode or from any other language that exposes the OSCLI interface.

25.5 Creating source files

Though the Assembler does not contain any editing facilities, it will accept files produced on just about any word processor or source editor.

The source you create should contain only printable characters, spaces and TAB characters, and each line should end with a carriage return byte (with optionally a line feed character before or after it).

The Assembler will treat the character used in saved Wordwise files to indicate a TAB (ASCII code &DD) as a "real" TAB character.

The Assembler will normally ignore any other characters in the source. Some word processors, though, may use characters with ASCII codes in the range 128 ... 255 for special purposes: the Assembler can be configured to change its action on encountering these.

This is described in detail in Customising the assembler[3].

3: Customising the assembler Section 25.20 on page 557

25.5.1 Using WordWise

Files saved by menu option 1 can be used directly as input to the Assembler, provided that you have not included any embedded commands.

The WordWise TAB character, which appears on the editing screen as a right-pointing arrow, will be accepted as a TAB by the Assembler.

Remember to press RETURN at the end of each source line.

Alternatively, you can spool the file to disc first using menu option 8 - but this will create a bigger file, that will take longer to assemble.

You may need some caution here if you are using the "|" character within string constants: remember that this is the WordWise "pad" character, so by default will appear as a space in the spooled file.

To overcome this, use the WordWise "pc" directive to redefine the pad character to something else.

25.5.2 Using View

A file created by the View SAVE, WRITE or EDIT commands can be directly input to the Assembler.

You should not include rulers, stored commands or highlight codes in this file.

You should always remember to press RETURN at the end of every source line: it will be necessary to turn off the Format and Justify options when typing the source, otherwise View will interfere with this.

25.6 Starting an assembly

The Assembler is started with the ***ASSEMBLE** command. This is followed by a number of "option flags" that tell the Assembler exactly how you wish it to operate.

The simplest use of the command would be, for example:

```
*ASSEMBLE SOURCE
```

where the source file name is SOURCE.

This format performs a "syntax check run" - no object code is produced. This will give you the fastest possibly assembly, and it can be very useful when checking a large program for errors.

To produce an object file containing the assembled code, you need to specify a -O option:

```
*ASSEMBLE SOURCE-OPROG
```

where the -O is followed by the name of the object file you wish to produce.

After assembly, the Assembler will return you to the language you were running when you issued the ***ASSEMBLE** command.

If you have a machine with shadow screen capability, the Assembler will automatically turn it on to give you the maximum space for symbols. It will return the shadow screen to its initial state when it finishes, and since this involves clearing the screen in a mode change, it will pause for you to press a key before it does so.

The command line options

The various options you can specify to the command are listed below. Each option flag consists of a "-" character followed by a letter, for example -S. Some option flags must be followed immediately by numbers or filenames. You must separate one option from another by spaces.

Options can be given in any order, but the first parameter to the command must always be the name of the first (or only) source file.

-A It makes the syntax of labels ADE[4]-compatible, and treats only the first 6 characters of each as significant.

This option implies a -R.

By default, all the characters of a label are significant.

-B Specifies that a memory buffer[5] is to be used for object code. This must be followed by a number in the range 1 ... 16

By default, no object buffer is used, and bytes are written to the object file one by one.

-C Specifies the default code listing level.

See the **CLST**[6] directive.

This must be followed by 0, 1 or 2.

By default, the Assembler will start an assembly with a code listing level of 1.

4: You must specify this option to assemble the source for **BBC KERMIT**, and it is recommended for sources originally written for the **ADE Assembler**.

5: memory buffer
Subsection 25.6.2 on page 458

6: **CLST**
Subsection 25.17.13 on page 533

Note that if you specify both -P and -F options, only the -F will take effect.

-F Specifies the name of a listing file, which must follow the option flag. If you give this option, the assembly listing will be written to the file instead of being sent to the screen.

If you do not use either the -P or -F options, the listing will be sent to the screen only.

-G Specifies that the Assembler is to restart the calling language[7] immediately the assembly is over.

By default, the Assembler pauses after assembly unless it is called from BASIC or its own CLI mode.

-L Specifies the default source listing level.

See the **LST**[8] directive.

This must be followed by 0, 1, 2 or 3.

You can also control the list level during the actual assembly with function keys f0, f1, f2 and f3.

By default, the Assembler will start an assembly with a source listing level of 0.

-M Instructs the Assembler to change screen mode before it starts to process the source. The number of the mode you wish to use should follow the letter M, and must be in the range 0 ... 7.

By default, the Assembler will leave the screen mode unchanged.

-O Specifies the name of the object file, which must follow the option flag. If you omit this, the Assembler performs a syntax check only and does not produce an object file.

7: language Subsection 25.6.3 on page 458

8: **LST** Subsection 25.17.45 on page 544

Note that if you specify both -P and -F options, only the -F will take effect.

-P Specifies that the listing is to be sent to a printer only.

If you do not use either the -P or -F options, the listing will be sent to the screen only.

-R If this is specified, 65C02 opcodes and addressing modes are rejected.

If you omit it, all 65C02 codes are assembled.

-S Specifies that lines skipped in conditionals are not listed.

See the **SFCOND**[9] and **LFCOND**[10] directives.

By default, the Assembler will list all skipped lines in conditionals.

-W Specifies that the Assembler should wait after displaying a line containing an error, so you can note down the details.

To resume assembly, press any key.

This option has no effect if you are sending output to a printer or to a file.

You can change the setting of this option at any time during assembly by pressing CTRL W .

By default, the Assembler will report all errors, but will not pause.

9: SFCOND
Subsection 25.17.57 on page 548

10: LFCOND
Subsection 25.17.43 on page 543

As an example, the line:

```
*ASSEMBLE SOURCE-OPROG-P -L 2 -B 8
```

will assemble from file SOURCE, putting the object code into file PROG.

The default listing level used will be 2, and the listing will be sent to the printer only.

An 8 kilobyte memory buffer is to be used for the object code to speed up assembly.

The Assembler will leave the screen in whatever mode you have selected, so that for large assemblies you may need to change to mode 7 to provide room in memory for the symbol table.

Shadow mode will automatically be selected if it is available, so that on the *B+*, *B+128* and *Master series* you will always have the maximum space available.

Alternative syntax forms for the command line

The example given above, and the output the Assembler will produce if you type

```
*HELP ASSEMBLER
```

detail the full form of the command line syntax.

As you gain experience in using the Assembler, however, you might wish to take advantage of various ways of specifying the options in a more compact form.

Firstly, the Assembler allows you to specify the options all at once, without the need to put the "-" flag before each of them.

Thus, you could type either

```
*ASSEMBLE SOURCE-M 3 -W -L 2
```

or the shorter, but less readable

```
*ASSEMBLE SOURCE-M 3WL 2
```

Secondly, the -O flag to specify an object file is optional.

The Assembler will take the second parameter of the command as an object file name, unless it starts with a "-" character.

Thus

```
*ASSEMBLE SOURCE-OPROG
```

and

```
*ASSEMBLE SOURCEPROG
```

are identical in effect.

If you should ever need to start the object file with a "-" character (*e.g.* if you wish to use a temporary file system on a Master 128) you will have to specify the -O flag explicitly.

25.6.1 Assembly on a second processor

The 65C02 Assembler is compatible with both the external 6502 second processor, and with the Master Turbo-Card. Using a second processor will give a faster assembly due to the higher clock rate of the processor, and will also give you more memory for the symbol table.

A second processor allows you to organise your source files to take advantage of a source file memory buffer. Any source file that is less than 14 kilobytes in size is ***LOAD** ed into memory and assembled from there: since the Assembler does not then need to wait for data to be fetched from disc, assembly will speed up dramatically. It is well worth arranging your program so each of its source files will fit into this buffer area.

The source buffer is automatically switched on when a second processor is in use.

25.6.2 The object code buffer

The Assembler allows you to select an object code buffer in memory. Code bytes generated from the source are written into this buffer, rather than being sent directly to the object file, and this gives a substantial increase in speed of assembly. Whenever the buffer is full, the assembly will pause in reading source to write the contents to the object file. It then resumes assembly until the buffer again fills, and so on.

You can specify the buffer size with the -B option. The flag is followed by a number between 1 and 16, giving the buffer size in kilobytes. Note that using a buffer means that there is correspondingly less space available for the symbol table.

The fastest assembly will be achieved when the object code fits entirely into the buffer. In this case, the Assembler will produce the object file with ***SAVE** at the very end of an assembly, which is substantially faster even than writing it in large blocks.

25.6.3 Returning to the calling language

When an assembly finishes, the Assembler will always restart the language ROM that was in use when you issued the ***ASSEMBLE** command. If this language was **BASIC** or the Assembler's own CLI mode, there will be no difficulty: however, some languages such as WordWise or View immediately clear the screen when they start, so you may not have time to read the final lines that the Assembler displays on the screen.

To overcome this, the Assembler is able to pause when it ends until you press a key. It will do this whenever the previous language is not **BASIC** or the Assembler's CLI mode.

If you don't wish it to pause (you may for example not want to read the final screen, or the language may not

clear the screen), simply specify a -G option in the command line. Now the Assembler will immediately restart the previous language without a pause.

There are two cases where the Assembler will always pause regardless of the -G option or the previous language. These are:

- When an error is detected in the ***ASSEMBLE** command you typed
- On a machine with a shadow screen that was turned off when you began assembly. The Assembler will turn the shadow screen on automatically, and will always turn it off at the end: as this requires a mode change which will clear the screen, it will always pause.

25.6.4 Use of control keys during assembly

During an assembly you can use various control keys to check on progress or give commands to the Assembler.

f0 **f1** **f2** and **f3** will force the Assembler to change the source listing level to 0, 1, 2 or 3 respectively.

Once you have pressed one of these keys the level you set is locked: the Assembler will process **LST**[11] directives in the source, but will not implement them.

This lets you, for example, change the list level during an assembly to check on what's happening, or to suppress a listing you decide you don't want.

To cancel the locking of the list level and return to that set by the last **LST** directive processed, simply press **CTRL L** .

11: **LST** Subsection 25.17.45 on page 544

CTRL H displays some help information to remind you about the use of control keys. The Assembler will pause after displaying the information: to resume simply press any key.

CTRL N **CTRL O** turn the BBC Computer's paged scrolling mode on and off respectively. You could use these keys, for example, to pause a listing on the screen after every screenful at some point during an assembly.

CTRL P controls whether the Assembler sends output to a printer. If it is printing, pressing **CTRL P** will stop it: if it is not printing, **CTRL P** will start it sending output to the printer.

If you have used the -F option to send output to a listing file, **CTRL P** can be used to switch output between the screen and the file, but will not affect any printer you may have connected to your machine.

CTRL Q lets you find out where the Assembler is in the current file. It will tell you whether it is on pass 1 or 2, then give you the number of the line it is about to process. If the number is followed by "(M)", the source line is a macro call and the Assembler is currently expanding that macro.

CTRL W reverses the current "wait after error" action set by the command line's -W option.

SPACE pauses the assembly. You can use this to examine a listing on screen at your leisure. To resume assembly, simply press any key.

25.7 Source Program Format

The Assembler will read source from files produced by any editor or word-processor program, as described in section Section 25.5 on page 450.

The source is seen as a number of lines, each ending with a carriage return byte ($0D). Line feed characters are ignored, as are all other control characters and (unless you have used one of the configuration options detailed in Customising the assembler[12]) any with the most significant bit (bit 7) set.

Note lines terminate with CR (0x0D), so if importing from a remote system you will need to ensure it's in BBC format and not Unix or Windows

12: Customising the assembler Section 25.20 on page 557

The general format of a line is:

```
label opcode operand comment
```

No line may exceed 132 bytes in length; any that do will be truncated.

Depending on circumstances, the label and operand fields may be optional or mandatory; or they may need to be omitted.

All lines may have a comment field, which is begun with a `;` or `\` character.

If you include a label field, it must begin in the first character of the line. You may separate the fields from each other with any number of spaces and/or TAB characters.

Any line whose first non-space character is `*` `;` `!` or `\` is treated as comment.

Lines beginning with `!` are treated specially if they occur within the definition of a macro, as explained in section **??** on page ??.

25.7.1 Syntax of the line

The label field

A label consists of a string of characters, starting with a letter, and containing any combination of letters, numbers and the characters `.` `$` and _.

Listing 25.1: Examples of valid labels

```
OSWRCH
osbyte
Program.start
ITEM_33:
```

Any lower-case letters are translated to upper-case.

Labels may be of any length, and all are significant unless you have specified the -A option in the command line, when only the first 6 are significant.

A label may be terminated with a `:` character that is not considered part of the label.

A label may be written in a line on its own: the value it is set to will be the value of the current address.

A variant of the label is the Sequence Symbol, described in Subsection 25.15.1 on page 509. This has the same format as a normal label, with the exception that the first character must be a `%` character.

Where the label is a macro name (i.e. in the MACRO directive) it may not exceed 8 characters in length.

The opcode field

The opcode field, if present, is separated from the label by one or more spaces or TABs. If there is no label, the field must be preceded by at least one space or TAB.

The opcode field can contain either a normal opcode mnemonic (such as LDA), an Assembler directive or pseudo-op (such as SFCOND) or the name of a macro that you have defined previously.

The operand field

For some opcodes and directives an operand field may be supplied. This will consist of one or more elements, separated by commas and optional spaces. Except within strings delimited by the single- or double-quote characters, spaces are not significant in the operand field. Thus you can use them to make complex numeric expressions more readable: for example you could write

```
LDA TABLE+(1+FRED/3*(BERT+\$1E))
```

as

```
LDA TABLE + (1 + FRED/3 * (BERT + \$1E) )
```

if you wished.

The comment field

The comment field can be used to annotate the line. It is begun with a `;` or `\` character: anything following is not processed by the Assembler.

25.7.2 The syntax of expressions

Throughout this chapter you will see references to "expressions", for example in the definitions of directives. The syntax of these is as follows:

Numeric expressions

These are indicated in definitions as *<expr>*, and consist of terms separated by arithmetic operators. For example, a valid expression might be

```
FRED*(2+BERT)
```

The Assembler performs all calculations using 16-bit arithmetic. Any results or intermediate values that overflow will be truncated, and no warning will be given.

The various elements that you can use in numeric expressions are these:

- Symbols
  These are the labels that you define in the label fields of lines, and in the evaluation of the expression they are replaced by the value of the label.
- Decimal constants
  e.g. 123
  These are numbers, composed of the characters '0'...'9'.
- Hexadecimal constants
  e.g. $AF34
  These are numbers, composed of hexadecimal digits '0'...'9' and 'A'...'F', preceded by a **$** or **&** character.
- Binary constants
  e.g. %11001001
  These are numbers composed of binary digits '0' and '1', preceded by a **%** character.

- Character constants
  These are single characters, enclosed in single-quotes. The Assembler will use the ASCII code of the particular character.
  You can specify that a character value is to be used with bit 7 set by preceding the character with a **^**, and can specify a control character by preceding it with a **|**. Thus the code for an **A** with bit 7 set is represented as **^A** and that for CTRL A as **|A**.
  The Assembler does not by default give you a means to specify that a character value is to be used as a control character and also with bit 7 set. You may find that this would be useful (if for example you wished to plant Teletext control codes such as $81 in a string), so the Assembler can be configured to permit this to be done.
  One of the options described in Customising the assembler[13]lets you set up the Assembler so that the **~** character (which appears on a Teletext screen as a "divide" sign) causes the next character to be output as a control character with bit 7 set. Thus, to obtain the hex code $81, you would write **~A**.
  Naturally, once you have selected this option, you will need to double the **~** character as well as the other two special characters should you wish it to appear itself in a string.
  Note that you must always include the final quote character.

e.g.'X'

Since the **|** character actually inverts bit 6 of the character that follows, you can obtain the code for "DEL" (hexadecimal code 0s7F) by writing **|?**

The character codes for **^** and **|** are obtained by writing the characters twice *i.e.* as **^^** and **||**

13: Customising the assembler Section 25.20 on page 557

- Operators
  A number of arithmetic operators can be used in expressions. They are divided into groups of varying priority, which, in decreasing order are shown in table Table 25.1 on the following page The = and

Table 25.1: Numeric operators

Op	Action
()	Parentheses
-	Unary minus
~	Unary 1's complement (NOT)
&	Bitwise AND
!	Bitwise OR
=	Equality
#	Inequality
>	Greater than
<	Less than
*	Multiplication
/	Integer division
+	Addition
-	Subtraction

operators return values of -1 and 0 for TRUE and FALSE respectively.
Additionally, expressions may be prefixed with two unary operators > and <, which select the low-byte and high-byte of the result respectively.
These operators have the lowest priority of all.
Thus, for example, suppose you have defined a table starting at label TABLE. To load the address of the table into X (the low byte) and Y (the high byte), you would write

The **MOD** and **DIV** operators used in the **BBC BASIC** assembler to achieve this result cannot be used

```
1    LDX \#>TABLE    ; load low byte
2    LDY \#<TABLE    ; load high byte
```

- Current address
  The current value of the address counter may be represented in expressions by the * character.

e.g. `*-3`

Strings

These are indicated in definitions as *<string>*, and consist of one or more ASCII characters, enclosed in delimiters. Only the single- and double-quote characters `'` and `"` can be used as delimiters, and the start and end delimiter must be the same.

For example:

```
'This is a string'
"And so, you see, is this"
```

To include the `'` or `"` character in a string, use the other as the delimiter.

Within a string you can obtain control characters and characters with B7 set in the way described above for character constants.

String expressions

These are indicated in definitions as *<stringexpr>*, and all involve comparisons between strings.

The general format is

```
<string> <op> <string>
```

where <op> is one of the operators in table Table 25.2 on the next page

Table 25.2: String operators

Op	Action
=	Equality
#	Inequality
>	Greater than
<	Less than

Page intentionally blank

25.8 The Output Listing

The assembly listing is produced in pass 2. You can control whether it appears or not, whether it goes to screen or printer, and the amount of detail it contains.

25.8.1 The source listing level

The main control you have over the listing is through the "source listing level". This is a value in the range 0 ... 3 that you can set either with the **LST**[14] directive, the -L command line option, or with function keys f0 , f2 or f3 during the course of an assembly.

14: **LST** Subsection 25.17.45 on page 544

The listing levels have the following meaning:

0 This level suppresses all listing except the reporting of errors.
1 This level will list all source lines that originate in source files, but not the expansions of macros.
2 This level lists all lines from source files and macro expansions, but not Macro Programming Language statements such as AIF.
3 This level lists all source lines.

By default, the Assembler sets the source list level to 0, so all you will see will be error reports. You can change the default setting in the ***ASSEMBLE** command by using the -L option: thus

```
*ASSEMBLE SOURCE-L2
```

would cause the Assembler to start with a default listing level of 2.

Within the source, you can control the list level with the LST directive. Thus:

```
LST 1
```

would set the list level to 1.

Simply writing

with no operand field will reset the value to the default, without you needing to build this value into the actual source.

Additionally, you can override the list level in use while an assembly is in progress. Pressing one of function keys 0 to 3 will force the corresponding list level to be adopted, so that, for example, you could force the Assembler to show you parts of an assembly to monitor what was being done. Pressing one of these keys will ”lock” the list level to that selected, so that LST directives in the source will not be able to turn the listing off again. The directives are noted, though, and you can return listing level control to them at any time by pressing CTRL L.

25.8.2 Source listing format

The listing begins with the source lines, including any code bytes they have generated.

The first character on the line indicates where the line was read from. If it came from an INCLUDE file or a CODE file the character will be ”I”; if the line is from a macro it will be ”M”. Otherwise, the character is a space.

The next element is the line number, reset to 1 at the start of each source file.

Next comes the address field. This will normally contain the current value of the address counter; in some directives, though, such as LST, it contains the value of the operand expression.

Then comes a hexadecimal representation of the first 3 object bytes generated by the line; this is followed by the source line itself. By default, if the source line generated more than 3 object bytes, these will be listed on subsequent lines.

For example, a listing line might show:

```
I 23 AB34:C9 0A CMP \#10
```

where the source line is the 23rd line in an INCLUDE file and reads "CMP #10". The code bytes generated are $C9 and $0A, and they are planted starting at address $AB34.

When listing a source line, the Assembler will expand any TAB characters it encounters. By default, TAB positions are defined at characters 8, 16, 25, 33, 41, 49, 57, 65, 73, 81, 89, 97, 115 and 123, with the first character of the source line being numbered 1. The default TAB positions have been chosen to give a neat listing layout using labels of up to 6 characters in length and typical operand field sizes. However, the first few TABs are not positioned at regular intervals, so you may find that you cannot easily lay out tables within comments.

You can overcome any problems, and choose the layout most suitable for your source, with the TABS directive. This lets you redefine the TAB positions used: for example

```
TABS 9,17,25,33,41,49,57
```

would set TAB stops at regular positions, with each field being 8 characters wide. You can return to the default TAB stops at any time with the directive

```
TABS
```

with no operands.

The Assembler can be customised to change the default TAB positions to any value you require: this is described in Customising the assembler[15].

15: Customising the assembler Section 25.20 on page 557

25.8.3 Listing object code bytes

As mentioned above, the first three code bytes generated by a source line are listed on that line, and any others will appear on subsequent lines. This may produce a lot of output, particularly if you define lots of strings, so the Assembler gives you control on how much it lists.

The amount is controlled by the ”code listing level”. This is a value in the range 0 ... 2 that you can set either with the CLST directive or the -C command line option.

The code listing levels have the following meaning:

0 Only the first 3 bytes generated by a line are listed.
1 All bytes are listed from all lines except the CODE directive.
2 All bytes are listed, however they are generated.

By default, the Assembler sets the code list level to 1. You can change the default setting in the ***ASSEMBLE** command by using the -C option: thus

```
*ASSEMBLE SOURCE-C0
```

would cause the Assembler to start with a default code listing level of 0.

Within the source, you can control the code listing level with the CLST directive. Thus:

```
CLST 2
```

would set the list level to 2.

Simply writing

```
CLST
```

with no operand field will reset the value to the default, without you needing to build this value into the actual source.

25.8.4 Symbol table listing format

If the source listing level is not zero at the end of the assembly, the Assembler will list the symbol table, showing you the value of all the symbols defined in the source.

The symbols are listed in alphabetical order, and only the first 13 characters of each will be shown. The value of each symbol is shown in hexadecimal.

If the value is shown as "????", the symbol was used but was not defined. If the value is "****", the symbol was defined more than once. A "-" character after the value indicates that the symbol was defined, but was not used anywhere else in the source.

25.8.5 Error reports

The Assembler will list all lines that generate errors, regardless of the listing level. The errors are reported with self-explanatory text messages following the line.

Where appropriate, the Assembler will attempt to indicate where on the line the error occurred. The text you might see would look something like:

```
****** Err : Undeclared symbol at about
    character 19
```

This indicates that the erroneous symbol is at about the 19th character in the line, which starts at character 1. Remember, though, that the number refers to the source line: the listing line will have expanded any TAB characters you used, and the number will then not correspond to what you see on the page.

25.8.6 Output page control

The Assembler provides a number of directives that you can use to specify the exact format of the output page, and to let you tailor the listing to your exact needs.

Setting page dimensions

The PAGE directive defines the size of the page you are using. It takes two parameters, which are the total depth of the page in lines, and the width in characters you wish printed.

Thus,

```
PAGE 66,132
```

tells the Assembler that the page is 66 lines deep, and that you wish lines to contain a maximum of 132 characters. The Assembler will always allow a small gap at the end of the page to avoid the perforations, and will truncate any lines longer than the maximum width you specify.

The value of the width you define with PAGE becomes effective at once. The depth, though, is not used until the next page throw, so you would normally follow the directive with a TTL or a SKP H directive.

By default, the Assembler will print lines of 80 characters on paper that is 66 lines deep. It moves to the top of a new page by sending a Form Feed ($0C) character to the printer.

Setting the page title

The TTL directive lets you set up a page title that is printed on the top of every listing page. The title can be up to 20 characters long, and can contain any printable text.

For example,

```
'Screen Dumper'
```

will print "Screen Dumper" at the top of each page.

The TTL directive will cause a page throw to occur immediately, and the next page output will use the title it defines.

Setting the timestamp

If you are using a Master 128, the Assembler will automatically print the current date and time at the top of each output page, taking the values from the built-in real-time clock. The other BBC models do not contain clocks, so the TIME directive lets you simulate the effect by defining the string used.

The TIME directive allows you to define a string of up to 25 characters that will appear in the page header. The string can contain any printable characters: it is not constrained to be a date and time value, so you can use it as a sub-title if you wish.

For example, you could use

```
TIME 'Friday at 1030'
```

or

```
TIME 'First Module'
```

The directive is ignored on a Master 128, so there is no need to change a source that is being assembled on both a model B and a Master 128.

It is possible to customise the Assembler to use a real-time clock fitted to a machine other than a Master 128: for details see Customising the assembler[16].

16: Customising the assembler Section 25.20 on page 557

Page layout directives

There are a number of directives that let you lay the output page out in more detail, so you can delimit sections of source for easier inspection and configure the axact appearance of each line.

The REP directive provides a convenient way of separating sections of code in the listing. For example,

```
REP 80
```

will print a line of "*" characters in the listing for you (and takes much less space in the source file than that line itself would).

The CHR directive lets you change the character used to make the line, so that, for example you could select lines of "-" characters with

```
CHR -
```

The SKP directive lets you break the listing up for easier reading. For example,

```
SKP 5
```

would leave a 5 line space on the page. The format

```
SKP H
```

causes the Assembler to start a new page. Note that the SKP directive itself will never be listed unless it contains an error.

25.8.7 Controlling the output destination

By default, the Assembler sends the assembly listing to the screen, in whatever mode you have selected. You can, though, choose to direct the assembly listing to a printer; or to output it to a file.

Sending the listing to a printer

You can direct output to the currently-selected printer by specifying the -P option in the command line. Thus, for example,

```
*ASSEMBLE SOURCE-L2 -P
```

would assemble from source file SOURCE and send the listing, at level 2, to the printer only.

At any time during an assembly, you can change the output destination by pressing **CTRL P** . This will switch the output destination between printer and screen: you can do this as often as you wish.

Sending the listing to a file

You can direct the assembly listing to a file by using the -F option in the command line. Thus, for example,

```
*ASSEMBLE SOURCE-L1 -F:3.LISTING
```

will produce an assembly listing at level 1, and will send it to the file :3.LISTING only.

Output sent to a file is laid out exactly as if it were going to a real printer. It will be broken up into pages, with the page depth and width controlled by the PAGE directive. After assembly, you can for example print the file with:

```
<CTRL-B>
*TYPE :3.LISTING
<CTRL-C>
```

Any printer control codes that you may send with the SYSVDU directive are not stored in the file, so you can output the file to any printer without the need to edit it out unsuitable codes first.

Note that if you specify both -P and -F options, only the -F will take effect.

If you have used the -F option, the action of CTRL P during assembly changes slightly. You can now use it to switch output between the screen and the file, allowing you, for example, to suppress listing of some parts of a source. It has no effect on any printer that you may have fitted.

Printer control

You will often need to send a sequence of control bytes to your printer before starting an assembly (for example to set it into condensed print mode).

The assembler gives you three directives to accomplish this automatically: **SYSVDU** sends a control sequence on both passes. **SYSVDU2** and **SYSVDU1** do the same, but only on pass 1 or pass 2 respectively.

If you have not used the -P command line option to produce a printed listing, the directives will not send any output to it: there is no need to alter your source to suit the times when you are printing or not printing.

To output a control sequence, simply write the bytes as operands to one of the directives, exactly as you would do in **BASIC**'s **VDU** command. Thus

```
\lancSYSVDUi 1,27,1,\programString{'E'}
```

sends the character ESCAPE (whose ASCIIcode is 27_{10} decimal) followed by 'E' to the printer only, on pass 1 of the assembly.

Note that you do not need first to send a code of "2" to activate printer output: the assembler itself ensures the bytes go to the printer if one is in use. The "1" codes in the example above act exactly as in **BASIC**'s **VDU** command, and send the following character to the printer only.

You can if you wish use **SYSVDU** to write to both printer and screen: thus

```
SYSVDU 'A','B'
```

sends characters 'A' and 'B' to both. However, the DISP family of directives give you a much better means of outputting messages to the screen.

The directives do not send any bytes to a listing file that you may have selected with the -F command line option. This means that you can safely print a listing file on any type of printer without the need to edit out possibly inappropriate control sequences first (but, of course, you may need to set the printer up yourself before printing the listing).

25.9 Directives Defining Data and Constants

The Assembler contains a number of directives that allow you to define values in the object file. You can specify byte values, two-byte or word values, and character strings, and also set up symbols to hold constants.

25.9.1 Defining constants

Assigning symbols to various constant values is a considerable help in writing assembly-level programs, both in terms of making a source more readable and also in helping when you need to change it.

For example, you could refer to an Operating System routine by writing its address $FFEE every time, or you could declare a symbol OSWRCH to stand for that name.

You can declare symbols to use as constants with the EQU directive. In the example above, you would write

```
OSWRCH EQU \$FFEE
```

and then call the routine as, say

```
JSR OSWRCH
```

You can write any numeric expression in the operand field of the EQU directive, but the expression must not contain any forward references.

25.9.2 Byte and word directives

There are 6 directives that plant byte and word values. DFB plants bytes, and DFW and DFDB plant word values. For convenience, and compatibility with other assemblers, the directives can also be written as DB, DW and DDB respectively.

All the directives can be given one or more expressions in the operand field, separated by commas, and these expressions give the values of the bytes or words to be planted. If you specify a label in the label field of the line, it is set to the address of the first byte of the first value planted.

Byte directives

The directive DFB (or its equivalent DB) allows you to define single-byte values which are written directly into the object file. The operand field is one or more expressions, each of which should lie in the range 0 ... 255.

For example,

```
DFB 1,2,\$FF,FRED+1
```

will write 4 bytes to the object file: the values will be 1, 2, 255 and the value of FRED+1.

Word directives

The directives DFW (or its equivalent DW) and DFDB (or its equivalent DDB) allow you to define two-byte or word values which are written directly into the object file. The operand field of each is a list of one or more expressions, each of which should lie in the range 0 ... 65535.

The difference between the directives lies in the order in which the two bytes are written to the object file. The DFW directive writes the bytes in low-byte, high-byte order; the DFDB directive writes them in high-byte, low byte order.

Thus, for example,

```
DFW \$1234,\$ABCD
```

will output the bytes $34, $12, $CD, $AB to the object file in that order, but

```
DFDB \$1234,\$ABCD
```

will output the bytes $12, $34, $AB, $CD.

Using repeat counts

Occasionally, you will need to use the byte or word directives to plant a number of bytes or words containing the same value. For example, you might need to write

```
DFB 5,5,5,5,5,5,5
```

to plant 7 bytes containing the value 5.

For small numbers of repeats, this is no problem, but it can be onerous if, say, you needed 49 bytes containing 5.

The Assembler gives you a very convenient shorthand way of repeating values in all the byte and word directives. Every expression you use can be prefixed with a "repeat count", which tells the Assembler to plant the value itself more than once. The repeat count is specified in "[]" brackets: thus, to plant the 49 bytes mentioned above, you could simply write

```
DFB [49]5
```

Like the value itself, the repeat count can be any expression, but it must not contain forward references. Thus you could write

```
DFB [(COUNT+27)/2]\$FF
```

You can freely mix values with and without repeat counts in any of the byte and word directives.

A repeat count value of zero will cause no bytes at all to be written to the object file.

25.9.3 Character string directives

The four directives ASC, CASC, STR and CSTR allow you to plant character string values into the object file.

All the directives require you to specify a character string, enclosed in single- or double-quotes, in the operand field. If you include a label in the label field of the line, it is given the value of the address of the first byte planted.

The ASC and STR directives

These directives plant simple strings in the object file. ASC plants the string exactly as you specify it; STR plants the string, and then automatically plants a carriage-return byte (code $0D) immediately after it.

Thus

```
ASC 'ABC'
```

plants the bytes $41, $42, $43 in the object file, and

```
STR 'ABC'
```

plants the bytes $41, $42, $43, $0D.

The CASC and CSTR directives

These directives plant "counted strings" in the operand field. They are similar to ASC and STR, except that the bytes planted are preceded by a single byte giving the length of the string.

Thus, to use the same examples as above,

```
CASC 'ABC'
```

plants the bytes $03, $41, $42, $43 in the object file, and

```
CSTR 'ABC'
```

plants the bytes $04, $41, $42, $43, $0D.

Note that the count byte planted by CSTR includes the $0D byte that the directive adds itself.

Planting special characters

The strings you give as the operands to ASC, CASC, STR and CSTR must of course be made up of printable characters in the source file, but the Assembler allows you a way of specifying control characters and characters with the most-significant-bit (bit 7) set to 1.

To specify a control character, you should precede the character itself with a **|** character. Thus, if you wish to plant the value of CTRL A, the string should contain **|A**. This method is identical to the way in which you specify control characters in Operating System commands such as *KEY.

To specify a character with bit 7 (the most significant bit) set, precede it with a **^** character: thus, to plant an **A** with bit 7 set, specify it in the string as **^A**.

For example,

```
ASC 'A|AB\^{}B'
```

plants the bytes $41, $01, $42, $C2 in the object file.

Should you need to specify the "|" or "^" characters, you should double them in the string (i.e. write them as **||** and **^^**)

The Assembler does not by default provide a means to plant characters as control characters also with bit 7 set. However, you can use a configuration option described in Customising the assembler[17] to let you do this: selecting the option makes the **~** character plant the character following as a control character with bit 7 set.

17: Customising the assembler Section 25.20 on page 557

Thus

```
ASC "~A"
```

would plant the byte $81 (i.e. a **CTRL A** with bit 7 set).

25.9.4 The HEX directive

This directive is rather a hybrid of the DFB and ASC directives. It plants a series of byte values into the object file, but you specify the values as a string of hexadecimal digits.

For example,

```
HEX '1234ABCD'
```

plants the byte values $12, $34, $AB, $CD into the object file.

The string you supply must contain valid hexadecimal digits (i.e. '0'...'9' and 'A'...'F'). You can give the letters in either upper- or lower-case. If the string contains an odd number of characters, the Assembler will automatically add a '0' to make the final hexadecimal value complete.

25.9.5 The CODE directive

The code directive gives you a convenient way of including things like screen dumps or already-compiled relocatable subroutines into your programs.

Suppose, for example, you had prepared a mode 7 screen dump with some Teletext editor system, and wanted to use this as a banner page when your program started. One way of including this into the code would be to examine the dump and convert it into the appropriate DFB directives, planting each byte explicitly by hand. However, this might be a very time-consuming process, and would be very error-prone.

Using CODE, though, removes the need to translate the dump into DFB or similar directives. If, for example, the dump was in file BANNER, writing

```
CODE BANNER
```

would cause the Assembler to read it in and copy the bytes directly to the object file, without processing them in any way.

By default, the listing will show only the first 3 bytes included from the CODE file, since otherwise a considerable amount of paper might be used. If you do want to see all the bytes in the listing, use the CLST directive to set the code listing level to 2 before using the CODE directive itself.

You cannot use the CODE directive within a macro, or from an INCLUDE file.

25.10 Object File Control Directives

The Assembler contains a number of directives that control the object file. They let you define the actual address that the code generated will run at, map symbol definitions into workspace, and set the load and execution addresses for the final result.

25.10.1 Defining the current address

In almost all programs you assemble, you will need to tell the Assembler the actual address at which the resulting code is to run. It will then be able to assign the correct address values to labels.

You define the current address with the ORG directive. For example, you could write

```
ORG \$1900
```

before the first code line in the source. This would set the address value to $1900, and the Assembler would produce code based on that address.

You can specify the address in the operand field of the ORG directive as any expression you like: however, since the Assembler must know the value to use on pass 1, the expression cannot contain forward references.

Note that the ORG directive permits you to specify only a two-byte or 16 bit address value. Disc filing systems use a 32 bit address value to control which processor a file is to be loaded into, and to set the top two bytes with the Assembler you should use the LOAD and EXEC directives, described in section Subsection 25.10.5 on page 491.

You can use as many ORG directives as you wish within your program source, and you can move the current address value to anywhere you wish. However, you should

remember that the ORG directive has no effect on where in the object file bytes are written. Thus, the source lines

```
ORG \$1900
LDA \#1
ORG \$3000
LDA \#2
```

would set the address to $1900, and write the two bytes for the LDA instruction to the object file. It then sets the address to $3000 and writes the bytes for the second LDA instruction: however, the four bytes will be contiguous in the object file - the Assembler will not generate a gap. You could use this effect, for instance, to write an object file containing many sections of overlay code - each section would be assembled to run at the same address, but they would lie one after the other in the object file.

If you do require to leave a gap in the object file, you can use the DS directive to generate the required gap yourself (see section Subsection 25.10.4 on page 490). There is no way of making the Assembler move backwards in the object file.

25.10.2 Reading the current address

You can use the value of the current address counter within expressions by referring to it with the special symbol "*". This can be used exactly as any other symbol in expressions, so that, say, you could work out the size of a data table with

```
TABLE DFB 1
    DFB 2
    DFB 3
T.SIZE EQU *-TABLE
```

where T.SIZE will be set to the number of bytes defined in the preceding table.

25.10.3 Using dummy sections

A "dummy section" is a convenient way of laying out symbols within the workspace that your program needs. For example, suppose your program needed some page zero workspace. You could build the actual numeric addresses into the source, but it's much better programming practice to define some symbols to identify the locations.

One way of doing this would be to use the EQU directive, which simply sets up a symbol and gives it a value. Thus you could write

```
WORK0 EQU \$00
WORK1 EQU \$01
WORK2 EQU \$02
```

to set up three symbols. This is a perfectly adequate technique, but suffers from the disadvantage that you would have to change a lot of the source if, say, you ever wanted the page zero workspace to start at some place other than address $00.

A better technique is to define workspace in a dummy section, which is a region of the source between DSECT and DEND directives. DSECT instructs the Assembler that a layout is being defined: it will process anything that follows it, but it won't generate any code for the object file. All it will do is work out what addresses any labels will take.

Thus, you could write the above example as

```
    DSECT
WORK0 DFB 0
WORK1 DFB 0
WORK2 DFB 0
    DEND
```

Here each of the workspace locations has been defined as one byte long with a DFB directive. Since the Assembler generates no code within a DSECT, the actual value you place in the operand field is quite arbitrary. This technique also lets you see clearly that you've set the locations up as one-byte values. If you wished, say, to make

them two-byte values, you could change the DFB directives to DW. The Assembler would then allow two bytes to each label, and the source would show this clearly.

In the example above, we have implicitly taken advantage of how the Assembler handles the address value within a **DSECT** ...**DEND** block. When it meets the first DSECT in a source, the Assembler makes a note of the address value it is using in the code, and resets it to zero for the **DSECT** ...**DEND** block. When it meets the DEND, it resumes using the address it saved at the start.

On the second occurrence of a DSECT, the effect is the same, except that now the address within the **DSECT** ...**DEND** block resumes at the value it reached in the last block.

If you don't wish to operate in this way, you can set the address value with the **DSECT** ...**DEND** block yourself with the ORG directive. Thus, if you wanted your page zero workspace to begin at some other location, you could write the source as

```
    DSECT
    ORG \$70
WORK0 DFB 0
WORK1 DFB 0
WORK2 DFB 0
    DEND
```

25.10.4 The DS directive

Quite often you will need to define a large area of workspace within a **DSECT** ...**DEND** block, for example as a source line input buffer. The DFB and DW directives are convenient ways of laying out byte and word values, but there are no built-in directives to lay out, say, a 256-byte buffer.

To do this, you can use the DS directive, which lets you lay out any number of bytes. For example, to lay out a 256-byte buffer you could write

```
    DSECT
    ORG         \$A00
BUFFER DS   256
    DEND
```

The operand field of the DS directive tells the Assembler how much space you wish to lay out.

Normally the DS directive is used in **DSECT** ...**DEND** blocks, but you can also use it in the code area of a program as well. In this case, the Assembler will write the appropriate number of bytes containing the value zero to the object file.

25.10.5 Defining object load and execution addresses

Every machine-code program on a BBC Computer needs to have a load address and an execution address associated with it. These values are stored in the file's catalogue entry: when you then *RUN the file, the filing system will load it to memory starting at the load address, and will enter it with a JSR instruction going to the execution address.

The addresses also tell the filing system whether the program is to be run on a second processor, or on the BBC Computer itself. The filing system regards load and execution addresses as 4 byte (32 bit) values. The lower two bytes or word of each address represent an actual address in memory and the top two bytes or word of the address is a value that indicates which processor the file is to be loaded on. The Assembler can handle only two-byte values in its arithmetic, so it is necessary to set the two parts of each address separately.

Specifying the low word of the addresses

There are two ways of setting up the low words of the addresses when you assemble programs. By default, the

Assembler will set the low words of both the load and the execution addresses to be the value set by the first ORG directive that is not in a dummy section, and in many cases this is all you need do.

If, though, you require something else, you can use the LOAD and EXEC directives to set the low word of the load and execution addresses respectively. Thus, for example,

```
LOAD \$1900
EXEC START
```

would set the low word of the load address to be $1900, and the low word of the execution address to be whatever address the label START is defined as.

Specifying the processor

If the top word of the load address is $FFFF, the file is loaded into the BBC Computer's own memory, whether or not you are using a second processor. If the top word is 0, though, the file will be loaded into the second processor if you are using one.

By default, the Assembler produces object files with the top word of the addresses set to $FFFF, so the files will load to the BBC Computer's memory. The MSW directive, though, lets you change this value: thus

```
MSW 0
```

lets you change the address to 0. You should always include this directive if you write programs to run on a second processor.

25.11 Source File Control Directives

The Assembler provides a number of directives that allow you to control the source files which you are assembling. This lets you split your source up into more than one file.

25.11.1 Chaining source files

If you are writing a large program, you will find it convenient to split the source into a number of small files, rather than keeping it in one large one. This approach not only makes editing easier, but also helps you structure the source to reflect the organisation of the program. Additionally, if you have a second processor, keeping source files to less than 14 kilobytes in size will give a very considerable increase in assembly speed (see section Subsection 25.6.1 on page 457).

You can instruct the Assembler to assemble multiple files with the CHN directive. This tells it to close the current source file and open the one specified in the operand field.

Thus, you could write

```
CHN FILE2
```

as the last line in FILE1. The Assembler will close FILE1, and continue assembly with FILE2. Note that if the CHN directive is not the last line in a file, anything following it will be ignored.

You can use CHN as many times as you wish in an assembly.

25.11.2 Including one file into another

A different way of using multiple source files is to use the INCLUDE directive. This tells the Assembler to start assembling another source file; unlike CHN, however, the first file is not closed. When the Assembler comes to the end of the included file, it resumes in the original file at the line after the INCLUDE directive.

One use of this directive might be to include a file of standard definitions or routines into a source. You might then write

```
INCLUDE STDSUBS
```

to assemble the source of some standard routines.

Lines that result from assembling an included file are marked in the listing by having an 'I' in the first column.

Note that you cannot use an INCLUDE directive inside a file that you are including already. Nor can you use the CHN and CODE directives.

25.12 Conditional Assembly

The 65C02 Assembler has full conditional assembly facilities, allowing you easily to change the source being compiled.

25.12.1 Assembly conditional on expressions

Conditional assembly that depends on the value of an expression is achieved by using the IF, ELSE and FI directives. The most general form of construction is this:

```
IF <expr>
.
TRUE block
.
ELSE
.
FALSE block
.
FI
```

<expr> can be any numeric expression, as described in section Subsection 25.7.2 on page 464, but it must not contain any forward references. If <expr> is non-zero, the Assembler takes the condition as TRUE, and will assemble the lines in the TRUE block. On coming to the ELSE it will ignore all the subsequent lines, until it reaches the FI; here assembly will continue normally once more.

If <expr> is zero, the condition is FALSE. The Assembler will ignore the lines that follow until it reaches the ELSE; it then will assemble the lines in the FALSE block.

If you wish, you can omit the ELSE and the FALSE block of code, to form

```
IF <expr>
.
TRUE block
.
```

```
FI
```

Here, if the condition is FALSE, the Assembler will ignore all lines up to the FI, so no code is assembled in this case.

For example, you might have a symbol DEBUG.MODE defined at the very start of the source to show if debug code is to be assembled or not. This might be defined as

```
DEBUG.MODE EQU -1 ;use -1 for TRUE, 0 for FALSE
```

Then, throughout the rest of the source, you can put the debug code in an IF condition of the form

```
IF DEBUG.MODE
.
debug mode version
.
ELSE
.
non-debug mode version
.
FI
```

Depending on circumstances, you may wish to reverse the condition tested to include some code if debug mode is not selected. You can use the unary 1's complement or NOT operator here: thus

```
IF ~DEBUG.MODE ;read as "IF NOT DEBUG.MODE"
.
non-debug mode version
.
FI
```

includes code if DEBUG.MODE is set to 0.

The **IF** ...**ELSE** ...**FI** construction can also be written using the directives **DO** ...**ELSE** ...**FIN** (or with any combination of them)

25.12.2 Assembly conditional on the existence of symbols

Two variants of the **IF** directive, **IFDEF** and **IFNDEF**, allow you to test for the existence or otherwise of symbols. **IFDEF** is TRUE if the symbol in the operand field does exist; **IFNDEF** is TRUE if it does not.

Thus the example above might have been written as

```
F DEBUG.MODE

g mode version



debug mode version
```

where the mere existence of the symbol DEBUG.MODE would select the debug code, regardless of its value.

25.12.3 Nesting conditionals

The Assembler allows you to ”nest” conditionals. Within a nested condition, an ELSE directive will be associated with the immediately preceding IF, IFDEF or IFNDEF.

25.12.4 Listing conditionals

By default, the Assembler will list all the lines that it skips in a conditional, printing an ”S” in the address field.

To save paper, or make the listing clearer, you can suppress listing of skipped lines with the SFCOND directive, or by specifying the -S option in the command line. You can re-instate the listing of false conditional branches at any time with the LFCOND directive.

Page intentionally blank

25.13 Progress Reporting

The Assembler supports several directives that allow you to output progress reports to the screen and to a printer or a listing file to show the course of an assembly.

25.13.1 The DISP, DISP1 and DISP2 directives

These all display a message on the screen, and also send it to a printer or to a listing file. DISP displays the text on both passes; DISP1 and DISP2 display the text on pass 1 only and pass 2 only, respectively. You can use INFO as a synonym of DISP2 is you wish.

The use of the directives is the same: for example, you might write

```
DISP 'Assembling debug code'
```

to display

```
---- Assembling debug code
```

on the screen.

The message you display can contain printable text, and you can also include some control characters. You specify them in the same way as you would in any other string, by preceding them with a **|** character. The control characters you can use are:

- **|M** start a new line
- **|J** start a new line
- **|G** make a beep sound

For example

```
DISP 'Line 1|MLine 2|G'
```

displays a 2 line message and rings the bell.

Additionally, the text can contain numeric expressions that are evaluated as the message is output. For example,

this would let you output a message to report the size of a section of code.

Within the text, you can specify an expression in one of two ways.

```
%D(<expr>)
```

will evaluate <expr> and display the result in decimal, and

```
%X(<expr>)
```

will display it in hexadecimal.

You can write any numeric expression you wish within the parentheses, but it must not contain forward references.

For example, to report the size of a section of code between labels START.CODE and END.CODE, use:

```
DISP2 'Size of code is \%X(END.CODE-START.CODE)
    bytes'
```

25.13.2 The WAIT, WAIT1 and WAIT2 directives

These output messages in the same way as DISP, DISP1 and DISP2, but then suspend the assembly until you press a key on the keyboard.

You could thus use these directives to enable you to change source discs during an assembly.

25.13.3 The QUERY directive

This offers a convenient way of setting up conditions for conditional assembly.

QUERY operates only on pass 1 of the assembly. It displays a message on the screen in the same way as the directives above, then waits for you to type in a line. The line is treated as a numeric expression and evaluated, the

result being set as the value of the symbol specified in the line's label field. If the expression is invalid, or contains a forward reference, the question will be repeated.

For example, suppose that your source contains optional debug code, and you select this with IF conditions of the form

```
IF DEBUG.MODE
.
debug code
.
FI
```

The value of DEBUG.MODE would normally be -1 to select debug code, or 0 if it were not needed.

Rather than define DEBUG.MODE in the source, and thus have to edit the source every time you wished to change it, you can make the Assembler ask for the value at the start of an assembly. A line such as:

```
DEBUG.MODE QUERY 'Use debug mode'
```

will output the question

```
---- Use debug mode?
```

on pass 1. You can now type in

```
-1
```

to select debug code, or

```
0
```

to suppress it.

Since the line you type is an expression, and can contain symbols, you can make this more friendly by changing the source to include

```
YES EQU -1
NO EQU 0
```

before the QUERY directive. Now you can reply either "YES" or "NO" to the question: the Assembler will evaluate the line and take the value of the symbol you type.

25.13.4 The STOP directive

STOP provides a useful way of abandoning an assembly part-way through. It operates on pass 1, and like DISP1, it displays a text message. Afterwards, though, it aborts the assembly with the error message ”Stopped”

25.14 Writing Simple Macros

The Assembler contains a powerful macro facility, allowing you to write very complex and sophisticated macros. In this chapter, we shall examine the simpler aspects of macros.

A macro is a way of producing a number of Assembler source lines merely by specifying its name in the opcode field of a line. For example, your source might contain many occurrences of the lines:

```
LDA VAR
CLC
ADC \#2
STA VAR
```

You could shorten the source, and make it easier to follow, by replacing each occurrence with a macro call: you might set up a suitable macro with the statements

```
ADDTWO MACRO
LDA VAR
CLC
ADC \#2
STA VAR
ENDM
```

Then, instead of writing the four lines of code, simply write one line

```
ADDTWO
```

The Assembler will expand the macro definition, and will automatically generate the four lines for you.

25.14.1 Using macro parameters

Although this is useful in itself, the macro as shown above is rather limited. It couldn't, for instance, be used to add 2 to the variable COUNT instead of VAR.

In order to provide this sort of flexibility, you can pass information to macros as "parameters". You could re-write the macro above to read

```
ADDTWO MACRO
LDA @P1
CLC
ADC \#2
STA @P1
ENDM
```

Here we have replaced the occurrences of VAR by the string "@P1". Now, when the Assembler expands the macro, it will replace all occurrences of "@P1" with parameter number 1 of the macro call. To add 2 to VAR, you would then write

```
ADDTWO VAR
```

and to add 2 to COUNT, you would write

```
ADDTWO COUNT
```

You can supply up to 9 parameters when you call a macro, and they are indicated in the body of the macro by @P1, @P2, @P3 and so on. (Note that you can also specify the parameters as @1, @2, @3 and so on, omitting the "P". This is adequate for existing code; however, for new programs you should use the "@P1" form, as this is necessary when you come to use the Macro Programming Language)

25.14.2 Specifying macro parameters

You can specify anything you want as parameters to macros. A macro can have up to 9 parameters, separated by a comma and optional spaces.

A macro with 3 parameters could be called with lines like

```
CHECK 1,FRED,27
CHECK 1 , FRED , 27
```

and so on.

Normally, the Assembler will remove leading and trailing spaces from each parameter. If you require leading or trailing spaces, or if the parameter has to include a comma, you will need to specify it as a string, delimited by single- or double-quote characters. Thus, a macro call might look like

```
THING 'Here, is a comma'
```

and "@P1" will be replaced in the macro body with the characters

```
Here, is a comma
```

Note that the string delimiters are not taken as part of the parameter proper.

25.14.3 Nesting macros

You can call macros from within macros, up to a depth of 5. If you attempt to nest deeper than that the Assembler will flag an error.

25.14.4 Redefining opcodes and directives

The Assembler allows you to set up macros to redefine any opcode or directive. For example, you might want to redefine the JSR (Jump-to-Subroutine) opcode to automatically save the registers before entering the subroutine. You could do this by declaring a macro called JSR thus:

```
1 JSR MACRO
2 PHA
3 TXA
4 PHA
5 TYA
6 PHA
7 JSR @P1
8 ENDM
```

Now, whenever the Assembler comes across a line with JSR in the opcode field, it will expand the macro JSR rather than obeying the opcode. It will plant the code to save the registers, and then will come to the line

```
JSR @P1
```

in the macro.

Here, because it is already in a macro, the Assembler will not use the macro JSR. Instead it will assemble the opcode JSR, planting the code to enter the subroutine.

25.14.5 Labels within macros

Suppose you wish to write a macro that includes a branch of some sort. You might write the macro definition as:

```
1 THING MACRO
2 LDA @P1
3 BEQ ZERO
4 EOR \#\$FF
5 ZERO STA @P2
6 ENDM
```

The first time the macro is called, it plants the code bytes, and defines the label ZERO as the destination of the BEQ instruction. On a subsequent call, though, the macro will produce the same code, and will attempt to define the value of ZERO again. This of course will fail, since it already exists from the first macro call.

The Assembler provides a way round this problem, by giving you a way of generating unique labels. Every time a macro is called, the Assembler sets up what you can regard as a special parameter on your behalf, which contains a string that is different for every macro call. This string is substituted, in the same way as ordinary parameters, by writing ”@$MC” in the line.

Thus, the above macro could be changed to be:

```
THING MACRO
LDA @P1
BEQ ZERO@\$MC
EOR \#\$FF
ZERO@\$MC STA @P1
ENDM
```

Then, on the first macro call, every occurrence of ZERO@$MC might be changed to ZERO1X1. On the next call, they become ZERO2X1, so that there is no clash between the macros.

Page intentionally blank

25.15 The Macro Programming Language

A very powerful feature of the Assembler is its Macro Programming Language. This allows you considerable control in how macros are expanded - you can construct loops, manipulate macro parameters and perform several other functions that allow you to build macros of great power.

Although this facility is mostly intended for use within macros, many of its facilities can also be used outside macros to great effect, as this section will explain.

The Macro Programming Language's facilities build on two source language features known as Assembly Time Variables and Sequence Symbols.

25.15.1 Sequence Symbols

These are "place markers" within your source files or within macros that the Macro Programming Language uses in loops. Using directives such as AGO and AIF, you can make the Assembler move up or down within a file or a macro, letting you repeatedly assemble some parts of the source or totally omit others.

Sequence Symbols are very similar to the labels that are part of the source proper, and they can contain the same characters. To distinguish them, Sequence Symbols always begin within a "%" sign in the first character of the line. The Sequence Symbol should be the only thing on the line: if you do put anything else there the Assembler will ignore it.

To take an example of how Sequence Symbols could be used, suppose your source file contained the lines

```
AGO %SKIP
ASC 'These lines will never '
ASC 'get assembled'
```

```
%SKIP
ASC 'But this one will'
```

The Assembler will encounter the AGO directive, and will then ignore everything in the source file until it finds the Sequence Symbol %SKIP. It will then resume its normal processing.

Although this example will actually work, the technique isn't greatly useful, as ignoring source lines can be done just as easily with the **IF** ...**ELSE** ...**FI** construction. However, AGO (and the various conditional skips such as AIF) also allows you to go backwards in the source or macro - there is no other way of achieving this.

The Sequence Symbols used in any file or macro are quite independent of those in any other; thus you can safely use ones of the same name in every file and macro, if you wish.

25.15.2 Assembly Time Variables

Assembly Time Variables, or ATVs, are string variables that the Assembler itself uses while assembling your source files. As it reads the program source (either from a file or from the definition of a macro) the Assembler continually looks for references to ATVs. These are replaced - before the line is assembled - with the contents of the ATV, thus allowing you to vary the source that is actually processed.

You can use ATVs in many ways. For example, the first line of a source might set an ATV string to hold the version number of the program you are assembling; the Assembler will then automatically replace every reference to that name with the string. Some ATVs are created by the Assembler itself, and let you incorporate such things as the name of the source file into the source itself.

The main use, though, is in controlling loops within macros and within source files.

Creating Assembly Time Variables

ATVs have names similar to the variables that form part of the source proper, and you can manipulate them with various directives.

Local and Global ATVs There are two types of ATV: local and global ones.

- Global ATVs exist for the whole of the assembly, and can be used anywhere.
  They are created and manipulated with the ASTR, ASET, ASTRIP and ALEN directives, which you can use even inside macros - the ATVs they create will continue to exist after the macro finishes.
- Local ATVs are created and manipulated by the MSTR, MSET, MSTRIP and MLEN directives.
  These directives can only be used inside macros, and the ATVs thy create can be used only within the same macro - they cease to exist at the end of the macro expansion. You would use these, for example, in controlling loops within a macro.
  In fact, you have already seen local ATVs in use in section Subsection 25.14.1 on page 503 on "Simple Macros". Whenever you invoke a macro, the Assembler creates local ATVs with names P1, P2, P3 and so on, each holding one of the parameters you supplied on the macro call.
  For example, the source line

```
NAME ASET 'OSWRCH'
```

  will set up an ATV called NAME, whose contents are the string 'OSWRCH'. Since the ASET directive has been used, the ATV is global, and can be used anywhere in the assembly.
  You can create local and global ATVs of the same name if you wish. However, if you wish to refer to the local ATV, you should be careful to always use the "M" form of the directives. The "A" forms

will always create global ATVs, even if local ones of the same name already exist.

String and Numeric Values All ATVs are stored by the Assembler as printable character strings. However, in many cases you will find you use them as counters for controlling loops: to make this easy, the Assembler will automatically convert ATVs from numbers to strings whenever necessary.

The rule used is quite simple. When processing ASET or MSET directives the Assembler examines the first character of the operand field. If it is a string delimiter, the operand is taken as a string and is not evaluated. If it is any other character, the Assembler treats the operand as a numeric expression, evaluates it, and converts the result into a string for storage.

Thus, the line

```
COUNT ASET 15+3
```

will set up the ATV COUNT, containing the string ”18”, but

```
COUNT ASET '15+3'
```

sets up the string ”15+3”.

The operand can be any expression, provided it does not contain any forward references: thus

```
COUNT ASET ADDRESS+10
```

sets COUNT to hold the string form of the result of adding 10 to the address label ADDRESS.

If you use expressions that mix numeric and character literal values, some caution is needed to make sure that the Assembler does what you want: for example,

```
VALUE ASET 'A'+3
```

would produce an error, rather than adding 3 to the code for the character ’A’. This is because the Assembler, seeing the operand field beginning with a ”’” character, will

treat the operand field as a string and find that it is invalid. To overcome this, you may be able to re-order the operand expression: thus you could write the above example as

```
VALUE ASET 3+'A'
```

Now the Assembler will treat the operand field as a numeric expression, since it does not begin with a string delimiter.

In some cases, though (particularly in macros where you may not know beforehand what the expression may look like) it may not be possible to re-order things safely. Here you can simply put a dummy number in front of the expression: thus

```
VALUE ASET 0+'A'+'B'
```

would add the character values 'A' and 'B', the leading "0" character forcing the Assembler to treat the operand as a numeric expression.

The ALEN and MLEN directives similarly convert from number to string for you. Here the operand must be a string: for example,

```
SIZE ALEN 'A short string'
```

sets SIZE to hold the string "14" - the length of the operand string, converted from a number into a string.

String slicing The ASET and MSET directives also allow to you to "slice" strings, or extract characters from inside them.

You perform this by adding some parameters in the operand field: for example

```
SLICE ASET 'abcdefghij',3,2
```

will set SLICE to hold the string "cd". The second operand parameter specifies the position of the first character to be sliced out (the string starts at character 1), and the third parameter specifies the number of characters to be sliced.

If you omit the third parameter, exactly one character is sliced: thus

```
SLICE ASET 'abcdefghij',3
```

will set SLICE to hold the string ”c”.

Occasionally, string manipulations such as slicing may result in strings that have leading or trailing spaces, and you may not be able to tell beforehand how many. The ASTRIP and MSTRIP directives remove all leading and trailing spaces, so that

```
NEW ASTRIP ' abcd '
```

would set the string NEW to be ”abcd”.

Efficient use of memory The strings contained by ATVs are held by the Assembler in the symbol table, and so compete for memory with other symbols. You can change the contents of an ATV to a string of different length as often as you wish: however, every time the string becomes longer the Assembler will allocate a new block of memory for it. The previous block cannot be used again, so continually increasing the size of an ATV can be extremely wasteful of memory.

To overcome this, the ASTR and MSTR directives let you pre-declare a block of memory big enough to hold the maximum size string you will use. For example,

```
NAME ASTR 50
```

sets up a block long enough to hold a 50-byte string without the need to get more memory.

The minimum amount of memory the Assembler will allocate is 5 bytes: this is enough to hold the largest possible numeric value, so that loop counters will not cause memory wastage as their values grow - you need not use ASTR and MSTR for them.

Simple Substitution of Assembly Time Variables

Once you have set up ATVs, you can use them to create variable source lines. We have already come across this concept in section Subsection 25.14.1 on page 503 on "Simple Macros", in the discussion of macro parameter substitution. There, we saw that if the source of the macro contained the line

```
1 LDA \#@P1
```

the Assembler would replace "@P1" with whatever you had specified as the first parameter of the macro when you called it.

ATVs are substituted into source lines in exactly the same way, and as we saw, macro parameters are in fact local ATVs.

The rule the Assembler uses is quite simple: whenever it detects an "@" sign in the source (other than in a comment line) it expects an ATV name to follow. The "@" and the name itself are replaced by the contents of the ATV before the line is assembled.

For example, you might keep the version number of a program in a small, quickly-editable file that is INCLUDEd into the source at the start of an assembly. This might contain the line

```
VERSION ASET "3.45"
```

which sets the ATV VERSION to hold the string "3.45". Then, at any point in the source where you needed to insert the version number (such as in a screen message) you could write, for example

```
TITLE ASC "SuperGame @VERSION"
```

The Assembler would replace "@VERSION" with "3.45" before it assembled the line, so the string actually planted would be

"SuperGame 3.45"

This technique lets you keep the version number in one easily-alterable place, and lets the Assembler take care of putting the up-to-date value (or parts of the value) into the source at the appropriate places.

There are some more complex uses of ATV substitution, and we shall discuss these later in section Subsection 25.15.3 on page 517 on "Writing Complex Macros".

Some useful points to note on substitution are:

- If you want the "@" character to appear in the line the Assembler processes, your source must contain two of them. Thus, if the line you really want to assemble is

  ```
  ASC 'VAT @ 15%'
  ```
  write it in the source file as

  ```
  ASC 'VAT @@ 15%'
  ```
- Once the Assembler finds an "@" character in the source, it assumes that what follows is an ATV name. The end of this name is assumed to be the first non-alphanumeric character that it meets, or the end of the line. In almost all cases, this will cause no difficulty, but occasionally will, usually in complex macros.
  As an example, suppose you had declared at ATV called EXPR, holding the string "10+". A subsequent source line might then read

  ```
  LDA \#@EXPR3
  ```
  and the intention is for this to be transformed into

  ```
  LDA \#10+3
  ```
  In this case, though, the substitution will fail, as the Assembler will look for an ATV called EXPR3. To force it to do as you want, write the source line as

  ```
  LDA \#@EXPR/3
  ```
  The "/" character enables the Assembler to detect the end of the ATV name, so it will look for EXPR as it should. The "/" will be discarded, so that the resulting line will be

```
LDA \#10+3
```
as intended. If you need a "/" character in the resulting line, write it as "//".
There is another techniques you might use in these circumstances. The Assembler does not regard spaces as significant in numeric expressions, so that you could write

```
LDA \#@EXPR 3
```
to achieve the same result.

- No ATV substitution is performed in comment lines, in lines containing Sequence Symbols, or in the definition of macros (i.e. between the MACRO and ENDM directives). Apart from these, though, substitutions can be made at any place in a line - you can substitute for labels, opcodes, operands or any parts of them.

25.15.3 Writing Complex Macros

Programming macro loops

Mostly, you will use the Macro Programming Language to program macro loops, controlled by various conditions.

Simple loops controlled by counter The simplest form of loop is one which is executed a fixed number of times, and needs only a counter to control it.

As an example, suppose that we need a macro to plant a number of bytes containing $FF with the DFB directive, the number being specified by the first parameter. (There are much easier ways of doing this than with a macro, of course - this only shows a general technique).

The macro definition might then be:

```
1 PLANT MACRO
2 COUNT MSET 0
3 %LOOP
```

```
4 DFB \$FF
5 COUNT MSET @COUNT+1
6 AIF @COUNT<@P1,%LOOP
7 ENDM
```

To see how this works, we can examine each line in turn, assuming that the macro was called with a line

```
PLANT 7
```

Line 1 : This is the macro definition line.

Line 2 : This line sets up a local ATV called COUNT, and gives it a string value of ”0”.

Line 3 : This is a Sequence Symbol marking the top of the loop. Note that there is nothing else on the line with it.

Line 4 : This is the DFB line that plants the $FF byte required.

Line 5 : This line increments the value of COUNT. As the Assembler reads the line, it encounters ”@COUNT”, which it replaces with the current string value of the ATV. Thus the first time this line is encountered, the Assembler will generate

```
COUNT MSET 0+1
```

The second time, it generates

```
COUNT MSET 1+1
```

and so on.

Line 6 : This tests whether the Assembler is to loop round once more. As with line 4, the Assembler will replace ”@COUNT” with the current value of the ATV. ”@P1” is, of course, replaced by the first parameter of the macro. The first time round, the line processed is

```
AIF 0<7,%LOOP
```

which is true, so the Assembler skips backwards in the macro to line 4 and resumes processing from there.

Loops accessing macro parameters Another frequently-needed form of loop is one in which all the parameters of the macro are accessed in turn. Suppose, for example, you need to write a macro THINGS, whose parameters are all numbers. Each number is to be planted in a byte in the object file with a DFB directive: to make THINGS interesting, the number of parameters must be variable.

Such a macro is fairly simple to write, but uses an ATV substitution technique that can, at first sight, be somewhat daunting. If the job were simply to plant the value of parameter 1, the line in the macro that does it would simply be

```
DFB @P1
```

However, we need to access each parameter in turn: the Assembler must somehow be made to see "@P1" the first time round the loop, "@P2" in the second, and so on. Effectively, then, we need a substitution technique that lets us have a variable ATV name: i.e. one that first substitutes the number ("1", "2", "3" etc) then substitutes for the ATV name so formed.

The Assembler can do this easily, since ATV substitution operates in a hierarchic fashion. For example, suppose that a source line contains

```
@(P@COUNT)
```

somewhere.

On seeing the first "@" character, the Assembler prepares to substitute an ATV. It finds, though, that the next character is a "(", so that it now expects a bracketed expression rather than an ATV name. It notes where it has got to in the source line, and explores within the brackets.

Now, though, it stores the characters it finds (rather than passing them on to be assembled), and expects to end up with a valid ATV name by the time it gets to the ")" character. It notes the "P", then finds the second "@". This makes it try again to substitute an ATV, and this

time it finds a real ATV called COUNT, which we shall suppose contains the string ”1”. After ”COUNT” it finds the ”)” ending the bracketed expression; thus, within the brackets the string it has built is now ”P1”.

Having ended the bracketed expression, the Assembler goes back to where it was. The ”(P@COUNT)” has provided the string ”P1”, and this now is a valid ATV name. So it proceeds to substitute the value of ATV P1, the first macro parameter, as we intended.

To see how we might use this technique, we can consider a definition of THINGS:

```
1 THINGS MACRO
2 COUNT MSET 1
3 %LOOP
4 AIF '@(P@COUNT)'='',%ALL.DONE
5 DFB @(P@COUNT)
6 COUNT MSET @COUNT+1
7 AIF @COUNT<10,%LOOP
8 %ALL.DONE
9 ENDM
```

To see how this works, we can examine each line in turn.

Line 1 : This is the macro definition line.

Line 2 : This line sets up a local ATV called COUNT, and gives it a string value of ”1”.

Line 3 : This is a Sequence Symbol marking the top of the loop.

Line 4 : Since the number of parameters must be variable, we need to test whether we’ve processed them all. You can see the substitution technique discussed above in use here to check if the next parameter is null - any parameters that you don’t supply in a macro call are strings of zero size. If the parameter is null, the Assembler skips forwards in the macro until it gets to the Sequence

```
Symbol %ALL.DONE.
```

Line 5 : This is the DFB line that plants the current macro parameter as a byte.

Line 6 : This line increments the value of COUNT, as in the previous example.

Line 7 : This tests whether the Assembler is to loop round once more. The final macro parameter is P9, so once COUNT reaches 10 the macro is finished.

Line 8 : This is the Sequence Symbol that line 4 skips to if it finds a null parameter.

Thus, if the macro was called with a line

```
THINGS 1,\$FE,FRED-1
```

the loop would be traversed three times, and the lines actually assembled would be

```
DFB 1
DFB \$FE
DFB FRED-1
```

The technique of hierarchical substitution, though most often used to access macro parameters in turn, can be used in many other applications: you can nest substitutions to as deep as you are likely to need, so that you might write something as horrendous looking as

```
LDA \#@(XX@(COUNT@PTR)B)
```

if you really needed to.

Changing macro parameters

Macro parameters are in fact local ATVs with names P1, P2, P3 and so on. This means that you can change them within a macro as you wish.

One example of this might be to allow the use of default macro parameters (although section Subsection 25.15.3 on page 523 below shows an automatic way to do this). Suppose that a macro parameter should be a number, whose default value is 1. You could define it as:

```
1 TEST MACRO
2 AIF '@P1'#'',%NEXT
3 P1 MSET 1
4 %NEXT
5 LDA #@P1
6 ENDM
```

Within this example, we have:

1 The macro definition line.
2 This tests if parameter 1 has been supplied. If so, it will not be a null string, so the Assembler skips to `%NEXT`.
3 This line sets parameter 1 to the default value.
4 This is the Sequence Symbol skipped to if the parameter is not defaulted.
5 This line actually uses the parameter. It will assemble correctly even if the parameter was not given, since in that case line 3 will have set it up to be the default value.

Setting default macro parameters

The example above showed one way of establishing a default value of a macro parameter, but in fact the Assembler gives you an automatic and easy way of doing this with the DEFPARS directive.

The effect of this directive, which you can issue at any time inside a macro, is to set the values of any of the macro parameters P1, P2, P3 and so on, unless they are already defined.

For example, you might call a macro with a line

```
FRED 1,,3
```

where you have defined parameters 1 and 3, but not parameter 2. If the macro now executes, say,

```
DEFPARS 100,200,300
```

the Assembler will check the parameters in turn. Parameters 1 and 3 are already defined, so the "100" and "300" values are not used. Parameter 2, though, is not yet defined, so it is set to "200" from this point on.

If you wished, say, to establish a default for only some of the parameters of a macro, simply specify only those parameters in the DEFPARS directive and default the others. Thus

```
DEFPARS ,,,44
```

sets a default for parameter 4, but has no effect whatsoever on parameters 1, 2, 3, 5, 6, 7, 8 and 9.

Listing control for macros

If you write complex macros with lots of loops, you will find that the Assembler actually executes a large number of lines that just contain the AIF, MSET directives, etc. This can swamp the listing, and make it hard to see the actual directives that plant data or code (as well as using up a large amount of paper).

To overcome this, list level 2 will not list directives such as MSET, AIF, etc, unless they contain errors. In order to see all the lines of a macro expansion, use list level 3. (Note that outside macros, ASET and other similar directives will list at level 2).

Exiting a macro prematurely

You may sometimes need to exit a macro as a result of a test. Depending on circumstances, you may be able to use AIF or AGO to skip to the physical end of the macro; however, you can also use the MEXIT directive. This exits the macro immediately, wherever in the macro body it is encountered.

Comment lines within macros

You can include comment lines within the definitions of macros, just as in normal code. For example, a short macro might be defined as

```
TEST MACRO
*
* Adjust value in PTR0
*
INC PTR0
ENDM
```

The three comment lines will be stored in the macro definition, and (if the listing level is set to 2 or higher) will be listed whenever you call the macro. This lets you output comments to describe the actions of the macro.

The disadvantage of this technique is that the comment lines are stored in their entirety in memory when the macro is defined, giving you correspondingly less space for the symbol table.

When you come to writing complex macros making great use of ATV's and loops, you will find it most advantageous to include liberal amounts of comment. In order that these comments, which purely document how the macro works rather than the code it generates, shouldn't use memory, comment lines beginning with "!" are not stored. Thus, you could write a macro as

```
TEST MACRO
!
! See if first parameter is null
!
AIF '@P1'\#'',%NEXT
!
! It is null, so reset it to "1"
!
P1 MSET 1
%NEXT
!
! Parameter 1 now definitely has a value,
! so we can use it
```

```
14 !
15 LDA \#@P1
16 ENDM
```

The lines beginning "!" will appear in the listing when you define the macro, but will not be stored and will not appear when the macro is called.

25.15.4 System ATVs

The Assembler provides a number of read-only ATVs that you can substitute. They provide information on the Assembler and the environment that you can use to control assembly. Each system ATV name starts with a "$" character: they are

$CLST The current code listing level.
$DEFCLST The default code listing level.
$DEFLST The default listing level.
$FLEVEL "1" if the current file is an INCLUDE file; "0" if it is not.
$FS The number of the filing system in use, as returned by OSARGS with A=0, Y=0. The Acorn DFS returns "4", Econet returns "5" and ADFS returns "8".
$LINE The number of the current source line.
$LST The current listing level.
$MC The macro call counter, used to generate unique labels within macros (see section Subsection 25.14.5 on page 506)
$MLEVEL The current macro nesting level. If not in a macro, the value is "0".
$OBJECT The name of the current object file, which may include leading or trailing spaces.
$OS The version of the MOS in use, as returned by OSBYTE with A=0.
$OSHWM The primary OSHWM value of the machine running the Assembler.
$PLENGTH The current page depth used in the assembly listing, as set by the PAGE directive.
$PWIDTH The current line width of the asembly listing, as set by the PAGE directive.

$PRINTING Indicates whether the assembly listing is being sent to the screen or to a printer or file. The value returned is ”0” if the listing is being sent to the screen, and ”-1” if it is being sent to a printer or a file.

$SOURCE The name of the current source file, which may include leading or trailing spaces.

$TIME The currently-set timestamp string, which may include leading or trailing spaces.

$TTL The currently-set page title string, which may include leading or trailing spaces.

$VERSION The version of the Assembler in use. This is returned as a numeric string, so that version 1.60 sets the string to be ”160”.

For example, the line

```
ORG @\$OSHWM
```

could be used to set the base address of the code to the OSHWM value of the machine being used, without the need to know what that value was.

25.16 Supported Processors

For descriptions of the opcodes, see:

6502 The Advanced User Guide for the BBC Micro[4,5]
6502 and 65C02 Master Reference Manual Part 2[6]

Although they describe the BBC BASIC Assembler syntax which cannot be used with this Assembler.

The Assembler supports all the opcodes of the 6502 and 65C02 processor families, using standard MOSTEK mnemonics.

25.16.1 Supported Opcodes

The opcode CLR can be used as a synonym for STZ

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022)
[5] *Adv. User Guide, BBC Micro*, Bray (Cambridge Microcomputer Center Nov. 1983)
[6] *Master Series Ref. Manual Part 2*, (Mar. 1986)

ADC	AND	ASL	BCC	BCS	BEQ	BIT
BMI	BNE	BPL	BRK	BVC	BVS	CLC
CLD	CLI	CLV	CMP	CPX	CPY	DEC
DEX	DEY	EOR	INC	INX	INY	JMP
JSR	LDA	LDX	LDY	LSR	NOP	ORA
PHA	PHP	PLA	PLP	ROL	ROR	RTI
RTS	SBC	SEC	SED	SEI	STA	STX
STY	TAX	TAY	TSX	TXS	TYA	

Table 25.3: The 6502 compatible opcode mnemonics available

BRA	DEA	INA	PHX	PHY
PLX	PLY	STZ	TRB	TSB

Table 25.4: The 65C02-only mnemonics

25.16.2 Address modes

The addressing modes common to both the 6502 and 65C02 processors are:

Table 25.5: Supported 6502 Address modes

Mode	Syntax
Implied	op
Accumulator	op A or op
Immediate	op #expr8
Low byte	op #>expr
High byte	op #<expr
Zero-page	op expr8
Indexed by X	op expr8,X
Indexed by Y	op expr8,Y
Absolute	op expr16
Indexed by X	op expr16,X
Indexed by Y	op expr16,Y
Indirect pre-indexed	op (expr8,X)
Indirect post-indexed	op (expr8),Y
Absolute indirect	op (expr)

In these definitions,

expr8 represents an 8 bit number
expr16 a 16-bit number
expr a value that is either 8 or 16 bits long

The addressing modes usable with the 65C02 processor only are:

Table 25.6: Supported 65C02 Address modes

Mode	Syntax
Zero-page indexed	op (expr8)
Absolute indirect pre-indexed	op (expr16,X)

25.17 Directives

The Assembler supports a large number of directives, or pseudo-ops. This section gives a detailed definition of each, arranged in alphabetical order.

Directives follow the normal syntax of opcodes. All can be followed by comment fields started with a ";" character; however, not all directives may have labels.

In the specification, items enclosed in "[]" brackets are optional, and can be omitted.

25.17.1 AGO

Description Skip to a Sequence Symbol

Syntax **AGO** sequence

Example AGO %LOOP

25.17.2 AIF

Description Skip to a Sequence Symbol if a condition is true

If <condition> is false assembly continues in the next line.

Syntax **AIF** expr,sequence

AIF stringexpr,sequence

Example AIF 'FREDA'>'FRED',%LOOP

25.17.3 AIFDEF

Description Skips to a Sequence Symbol if a symbol has been defined.

Syntax **AIFDEF** symbol,sequence

Example AIFDEF RAM.CODE,%DOING.RAM

25.17.4 AIFNDEF

Description Skips to a Sequence Symbol if a symbol has not been defined.

The syntax is as for AIFDEF.

25.17.5 ALEN

Description Sets a global ATV to the length of the operand string.

Syntax label **ALEN** string

Example SIZE ALEN 'This is a string'

25.17.6 ASC

Description Defines an ASCII string.

Syntax [*label*] **ASC** string

Example TITLE ASC 'SuperGame 3.45'

Within string you can use the techniques of section Subsection 25.9.3 on page 484 to plant control characters or characters with bit 7 set.

If a label is specified, it is set to the address of the first byte of the string.

25.17.7 ASET

Description Sets a global ATV to contain a string.

Syntax label **ASET** string [,*expr1* [, *expr2*]]

label **ASET** expr

Example COUNT ASET 100

The final two parameters in the operand field allow the string parameter to be sliced.

Both are expressions that must not involve forward references.

expr1 specifies the first byte to be sliced from the string (the first byte in the string is byte 1).

expr2 defines the size of the slice; if omitted, one byte is extracted.

25.17.8 ASTR

Description Allocates space for a global ATV.

Syntax label **ASTR** expr

Example NAME ASTR 150

expr, which must not contain forward references, defines the space needed. The value can be up to 255 bytes.

25.17.9 ASTRIP

Description Sets a global ATV to a string value, removing leading and trailing spaces in the process.

Syntax label **ASTRIP** string

Example MIDDLE ASTRIP ' aardvark '

25.17.10 CASC

Description This is similar in action to ASC[18], but will plant a byte containing the number of characters within the string before the string itself.

18: ASC Subsection 25.17.6 on page 530

25.17.11 CHN

Description Closes the current source file and starts another.

Syntax [*label*] **CHN** filename

Example CHN :2.NEXTBIT

The directive is not allowed in an INCLUDE file.

25.17.12 CHR

Description Defines the character used by the REP[19] directive.

Syntax [*label*] **CHR** character

Example CHR #

The character is specified literally, and not as a character constant.

19: REP Subsection 25.17.56 on page 548

25.17.13 CLST

Description Sets the code listing level.

Syntax [*label*] **CLST** [*expr*]

Example CLST 2

<expr>, which must not contain forward references, gives the new listing level. If omitted, the level is reset to the default from the command line.

The values allowed are:

0 List only the first 3 bytes produced by any line.
1 List all the bytes planted by any line except a CODE directive.
2 List all the bytes planted by all lines.

25.17.14 CODE

Description Include a file (such as a screen dump) directly into the object file.

Syntax [*label*] **CODE** filename

Example CODE SCRDUMP

The specified file is copied, with no interpretation, directly to the object file.

If label is specified it is set to the address of the first byte copied.

The address count is increased by the size of the file.

25.17.15 CSTR

Description Similar to STR[20], but this directive plants a byte containing the number of characters in the string before the string itself.

The count includes the $0D byte that is automatically added at the end of the string.

25.17.16 DB

This is identical to DFB[21].

25.17.17 DDB

Description Defines a number of two-byte values, output in high-low order.

Syntax [*label*] **DDB** expr [, *expr* ...]

Example TABLE DDB 28,32105,[10]$FFFF

Each expression in the operand field is evaluated and stored as a two-byte value.

If label is present, it is set to the address of the high-order byte of the first expression.

Each expression can be preceded by a repeat count as described in Subsection 25.9.2 on page 482.

20: STR Subsection 25.17.60 on page 549

21: DFB Subsection 25.17.20 on page 535

25.17.18 DEFPARS

Description Sets up default parameters for a macro.

See section Subsection 25.15.3 on page 523 for full details of this directive.

25.17.19 DEND

Description Terminates a DSECT[22].

Syntax [*label*] **DEND**

Example DEND

The address that was saved when the DSECT directive was encountered is reset.

22: DSECT Subsection 25.17.29 on page 538

25.17.20 DFB

Description Defines a number of single-byte values.

Syntax [*label*] **DFB** expr [, *expr* ...]

Example TABLE DFB 0,[27]1,FRED+2

Each expression in the operand field is evaluated and stored as a byte value.

If label is present, it is set to the address of the first expression.

Each expression can be preceded by a repeat count as described in section Subsection 25.9.2 on page 482.

25.17.21 DFDB

This is identical to DDB[23].

23: DDB Subsection 25.17.17 on page 534

25.17.22 DFS

This is identical to DS[24].

24: DS Subsection 25.17.28 on page 537

25.17.23 DFW

This is identical to DW[25].

25: DW Subsection 25.17.30 on page 538

25.17.24 DISP

Description Displays a message on both pass 1 and pass 2.

Syntax [*label*] **DISP** string

Example DISP 'Program size is %D(*-$8000)'

string may contain references to variables: for details see section Subsection 25.13.1 on page 499.

25.17.25 DISP1

As for DISP, except that the message is displayed only on pass 1.

25.17.26 DISP2

As for DISP, except that the message is displayed only on pass 2.

25.17.27 DO

A synonym for IF[26].

26: IF Subsection 25.17.38 on page 541

25.17.28 DS

Description Reserves space in the object file, setting the bytes so affected to zero.

Syntax [*label*] **DS** expr

Example DS $C000-PROG.TOP.ADDRESS

expr, which must not include forward references, defines the amount of space to be reserved.

If label is present it is set to the address of the first byte reserved.

25.17.29 DSECT

Description Opens a "dummy section" that allows an area of memory to be defined without generating any object code.

Syntax [*label*] **DSECT**

Example DSECT

When the DSECT directive is processed, the current address is noted by the Assembler, and the value is reset to that at the end of the previous dummy section (or 0, if this is the first).

An ORG directive can be used to change the value if required.

When the dummy section is terminated by a DEND[27] directive the old value of the address is restored.

27: DEND
Subsection 25.17.19 on page 535

25.17.30 DW

Description Defines a number of two-byte values output in low-high order.

Syntax [*label*] **DW** expr [, *expr* ...]

Example TABLE DW FRED,BERT

Each expression in the operand field is evaluated and stored as a two-byte value.

If label is present, it is set to the address of the low-order byte of the first expression.

Each expression can be preceded by a repeat count as described in section Subsection 25.9.2 on page 482.

25.17.31 ELSE

Description Marks the end of the TRUE branch of a conditional assembly block begun by IF[28], IFDEF[29] and IFNDEF[30].

For details see section Section 25.12 on page 495.

25.17.32 ENDM

Description Terminates the definition of a macro begun by the MACRO[31] directive.

Syntax ENDM

25.17.33 EQU

Description Assigns a value to a symbol.

Syntax label **EQU** expr

Example OSWRCH EQU $FFEE

expr must not contain forward references.

28: IF Subsection 25.17.38 on page 541

29: IFDEF Subsection 25.17.39 on page 541

30: IFNDEF Subsection 25.17.40 on page 542

31: MACRO Subsection 25.17.46 on page 545

25.17.34 EXEC

Description Specifies the bottom 16 bits of the 32 bit execution address of the object code.

Syntax [*label*] **EXEC** expr

Example EXEC START+3

The catalogue entry of the object file will be updated to show the specified execution address.

expr can only be a 16-bit quantity: the value of the two top bytes of the address are set by the MSW[32] directive.

If this directive is not used, the execution address is set to the address defined by the first ORG that is not in a dummy section.

32: MSW
Subsection 25.17.52 on page 546

25.17.35 FI

Description Terminates a conditional assembly block begun by IF[33], IFDEF[34] and IFNDEF[35].

For details, see section Section 25.12 on page 495.

33: IF
Subsection 25.17.38 on page 541

34: IFDEF
Subsection 25.17.39 on page 541

35: IFNDEF
Subsection 25.17.40 on page 542

25.17.36 FIN

This is identical to FI.

25.17.37 HEX

Description Defines a series of bytes in the object program, the bytes being specified as a hexadecimal string.

Syntax [*label*] **HEX** string

Example BYTE.LIST HEX 'AB34FF2E7E'

string must contain hexadecimal digits (i.e. '0'...'9' or 'A'...'F'), and each pair of digits is output as one byte.

If label is present it is set to the address of the first byte from the string.

25.17.38 IF

Description Begins a conditional assembly block.

Syntax [*label*] **IF** expr

If expr, which cannot contain forward references, is non-zero, the condition is TRUE, otherwise it is FALSE.

For details, see section Subsection 25.12.1 on page 495.

25.17.39 IFDEF

Description Begins a conditional assembly block.

Syntax [*label*] **IFDEF** symbol

If symbol has been defined, the condition is TRUE, otherwise it is FALSE.

For details, see section Subsection 25.12.2 on page 497

25.17.40 IFNDEF

Description Begins a conditional assembly block.

Syntax [*label*] **IFNDEF** symbol

If symbol has not been defined, the condition is TRUE, otherwise it is FALSE.

For details, see section Subsection 25.12.2 on page 497.

25.17.41 INCLUDE

Description Begins assembly from a new source file, afterwards resuming with the current file.

Syntax [*label*] **INCLUDE** filename

Example INCLUDE :3.DEFINES

The Assembler will process lines from the specified file.

Lines in the listing file will begin with an ”I” to indicate their origin.

At end-of-file, it resumes after the INCLUDE directive.

The directive may not be used within an INCLUDE file or a macro.

25.17.42 INFO

This is identical to DISP2[36].

36: DISP2 Subsection 25.17.26 on page 537

25.17.43 LFCOND

Description Specifies that lines skipped in a conditional are to be listed.

Syntax [*label*] **LFCOND**

For details, see section Subsection 25.12.4 on page 497.

25.17.44 LOAD

Description Specifies the bottom 16 bits of the 32 bit load address of the

Syntax [*label*] **LOAD** expr

Example LOAD $1902

The catalogue entry of the object file will be updated to show the specified load address.

expr can only be a 16-bit quantity: the value of the two top bytes of the address are set by the MSW directive.

If this directive is not used, the load address is set to the address defined by the first ORG that is not in a dummy section.

25.17.45 LST

Description Sets the source listing level.

Syntax [*label*] **LST** expr

expr, which must not contain forward references, give the new level.

If omitted, the level is reset to the default specified in the command line.

The allowed values are:

0 No lines are listed.
1 List lines from files, but not from macros.
2 List all lines, but not directives such as AIF within macros.
3 List all lines.

Lines containing errors are always listed.

25.17.46 MACRO

Description Defines a macro.

Syntax label MACRO

Example THING MACRO

The following lines are stored as the definition of the macro.

No ATV substitution is performed on the lines.

The definition is terminated by an ENDM[37] directive.

Within the definition, comment lines starting with "!" will be listed but not stored in memory.

The macro name may not be more than 8 characters long.

The MACRO directive may not be used within a macro.

37: ENDM
Subsection 25.17.32 on page 539

25.17.47 MEXIT

Description Causes a macro to terminate immediately.

Syntax MEXIT

25.17.48 MLEN

As for ALEN[38], but this can be used only within a macro and creates a local ATV rather than a global one.

38: ALEN
Subsection 25.17.5 on page 530

25.17.49 MSET

As for ASET[39], but this can be used only within a macro and creates a local ATV rather than a global one.

39: ASET
Subsection 25.17.7 on page 530

25.17.50 MSTR

As for ASTR[40], but this can be used only within a macro and creates a local ATV rather than a global one.

40: ASTR
Subsection 25.17.8 on page 531

25.17.51 MSTRIP

As for ASTRIP[41], but this can be used only within a macro and creates a local ATV rather than a global one.

41: ASTRIP
Subsection 25.17.9 on page 531

25.17.52 MSW

Description Sets the two top bytes of the load and execution addresses.

Syntax [*label*] **MSW** expr

Example MSW 0

By default the bytes used are $FFFF, but if you are assembling code to run on a second processor you will need to change them to the appropriate value.

25.17.53 ORG

Description Sets the value of the address counter.

Syntax [*label*] **ORG** expr

Example ORG $1900

expr, which must not contain forward references, gives the new value.

The first ORG directive that is not in a DSECT[42]...DEND block also defines the default load and execution addresses.

42: DSECT
Subsection 25.17.29 on page 538

The object file is not affected in any way; thus if you use ORG to advance the address value, the Assembler will not plant any bytes in the object file to fill the gap.

To accomplish this effect, you can use the DS directive.

25.17.54 PAGE

Description Defines the page length and width of the listing.

Syntax [*label*] **PAGE** expr1 [,*expr2*]

Example PAGE 88,132

expr1 is the physical number of lines on the paper. The Assembler leaves a small gap at the end of each page, to avoid the perforations.

Default page length 66

expr2 is the number of characters to print on each line. If omitted, the value is set to 80.

Default Page Width 80

PAGE has no effect on the screen display.

25.17.55 QUERY

Description Displays a question and reads in a reply, which is evaluated as an expression and used to define a label.

Syntax [*label*] **QUERY** string 1

For details, see section Subsection 25.13.3 on page 500

25.17.56 REP

Description Outputs a comment line to the listing to mark a source division.

Syntax [*label*] **REP** [*expr*]

expr must not exceed 132 and must not contain forward references.

If you omit the value, the Assembler will fill the output line to its full width.

For details, see section **??** on page ??

25.17.57 SFCOND

Description Specifies that lines skipped in a conditional should not be listed.

Syntax [*label*] **SFCOND**

For details, see section Subsection 25.12.4 on page 497

25.17.58 SKP

Description Leaves a gap in the listing or starts a new page.

Syntax [*label*] **SKP** expr

[*label*] **SKP H**

Example SKP 5

In the first form, <expr> gives the number of lines to be left blank. In the second, a new page is started.

The SKP directive itself is not listed.

25.17.59 STOP

Description Aborts the assembly on pass 1, outputting a message.

Syntax [*label*] **STOP** string

Example STOP 'Too much in page zero'

The Assembler will display string, then abort with the error message "Stopped".

25.17.60 STR

Description This is similar to ASC[43], except that the Assembler will automatically add a $0D byte (Carriage Return) to the bytes generated.

43: ASC Subsection 25.17.6 on page 530

25.17.61 SYSCLI

Description Issues a MOS command, allowing you, for example, to change the default disc drive.

Syntax [*label*] **SYSCLI** string

Example SYSCLI 'DRIVE 2'

You need not begin the string to be issued as a MOS command with a "*", but it doesn't matter if you do.

Note that care is needed over the commands you try to execute.

Anything that uses memory will almost certainly corrupt the Assembler's workspace and cause assembly to go horribly wrong.

25.17.62 SYSVDU

Description Outputs one or more bytes to the screen and printer.

Syntax [*label*] **SYSVDU** expr [,*expr* ...]

Example SYSVDU 1,27,1,'E'

The directive is similar to the BASIC VDU command. The low-byte of each expr is written with a call to OSWRCH.

This gives you a means of sending control codes to a printer to set page formats, etc.

SYSVDU will send all the bytes you specify to the screen.

If you have used the "-P" command-line option, they will also be sent to the printer.

To send the bytes only to the printer, precede each with a "1" value.

The bytes are not sent to a listing file selected with the "-F" option.

SYSVDU operates on both passes of the Assembler.

25.17.63 SYSVDU1

This directive is identical to SYSVDU, but operates only on pass 1.

25.17.64 SYSVDU2

This directive is identical to SYSVDU, but operates only on pass 2.

25.17.65 TABS

Description This directive allows you to redefine the TAB positions the Assembler will use when listing source lines.

Syntax [*label*] **TABS** expr [,*expr* ...]

You may specify up to 14 TAB positions: any excess values will be ignored. Specifying the directive with no operand filed will reset the built-in defaults.

See section Subsection 25.8.2 on page 470 for more details.

25.17.66 TTL

Description Defines the title string on the top of each listing page.

Syntax [*label*] **TTL** string

Example TTL 'Screen Dumper'

The Assembler uses the first 20 bytes of the string as the page title, and also starts a new page.

25.17.67 TIME

Description Defines the timestamp string output on each listing page.

Syntax [*label*] **TIME** string

Example TIME '25-August-1986 13:00'

The Assembler uses the first 24 bytes of the string for the new timestamp.

The value need not be a time; you can use it as a subsidiary page title.

On a Master 128 the directive is ignored, as the timestamp is read from the real-time clock. The Assembler can also be configured to read a third-party real-time clock fitted to other models.

25.17.68 WAIT

This displays a string (see under DISP[44]), then pauses the assembly until you press a key.

44: DISP
Subsection 25.17.24 on page 536

25.17.69 WAIT1

As for WAIT, but operates only on pass 1.

25.17.70 WAIT2

As for WAIT, but operates only on pass 2.

25.18 Differences from the ADE Assembler

The 65C02 Assembler contains a number of similar directives to the ADE Assembler, and in almost all cases it will be possible to assemble programs written for ADE with it.

A few directives, though, have some minor differences, and there are also some small points on source format that must be considered.

- Only single- and double-quote characters may delimit strings.
- The second delimiter must not be omitted from character constants.
- Character constants obey the same rule as strings in the ASC and STR directives for control characters and those with bit 7 set. Thus, the "|" and "^" characters need to be doubled if they are being defined as constants.
- Strings containing commas or leading or trailing spaces are specified as macro parameters within normal string delimiters rather than in [] brackets.
- Macro parameters must always be separated by commas.
- The SYSFX directive is ignored: to achieve the desired effect you should use the SYSCLI directive to issue a *FX command.
- Macro libraries are not implemented.
- Local labels within macros are not implemented. The system ATV $MC should be used to construct unique labels as described in section Subsection 25.14.5 on page 506.
- Expressions are evaluated fully hierarchically rather than left-to-right.
- The "#" arithmetic operator is used for "inequality", returning -1 if unequal and 0 if equal, rather than "modulus".

- The LST directive has a numeric operand rather than the strings OFF, ON and FULL. For ADE compatibility, define symbols OFF, ON and FULL with the EQU directive to have values 0, 1 and 2. List level 3 has no equivalent in ADE.
- The load and execution addresses are set by the first ORG directive only. To set them to different values use the LOAD and EXEC directives.

25.19 Differences from previous releases

Version 1.60 of the 65C02 Assembler includes several new facilities and a number of corrections from the previous 1.52 release.

The most important changes made are as follows:

- A new directive, TABS, allows the layout of listing lines to be configured. A patch facility allows the default settings to be configured.
- New directives SYSVDU1 and SYSVDU2 perform the same action as SYSVDU but only on pass 1 and pass 2 respectively.
- Comment lines can be started with a "!" character; such lines within macro definitions are not stored in memory and can be used to annotate the macro itself.
- Printer handling has been extensively rewritten to avoid problems when using Econet print servers.
- Assembly listings can be directed to a file with the "-F" command-line option.
- String slicing operations in the ASET and MSET directives now work exactly as documented. In version 1.52, the string start and slice size parameters were reversed, and the first byte of the string was numbered 0 and not 1.

- A new MOS command, *CLI, allows the Assembler's CLI mode to be explicitly entered from any language.
- A REP directive with no operand now fills the output line to its defined width.
- System ATVs $PWIDTH and $PLENGTH allow the values set by the PAGE directive to be read.
- System ATV $PRINTING allows the source to determine if an output listing is being sent to printer or file, or to the screen only.
- A repeat count of zero in directives such as DB causes no bytes to be written to the object file, rather than repeating the operand 65536 times.
- A zero operand in the DS directive causes a gap size of zero rather than 65536.
- Patch facilities allow the Assembler's translation of source bytes in the range 128..255 to be configured.
- Patch facilities force the Assembler to read a real-time clock on machines other than a Master 128.
- Patch facilities enable the " " character to be used in character constants and strings to plant the following character as a control character with bit 7 set.
- Patch facilities allow the Form Feeds sent to a printer at the end of a listing to be suppressed.
- Source file repositioning (in directives such as AGO, AIF) no longer causes random failures with files larger than 14kbytes on a second processor.
- The DB directive now accepts negative operands in the range -1 to -127.

25.20 Customising the assembler

This is Appendix 5 in the original documentation.

However this feels like it should be part of the main content hence it's presented here rather than in the Appendices section.

Version 1.60 of the Assembler contains a number of options that can be configured or activated by changes to the ROM image. Typically, the facilities controlled in this way are highly specific to particular hardware or modes of working, rather than of general use.

In order to configure the Assembler you will need to alter some bytes within it, using a machine code monitor such as Exmon or Beebmon. The things you need to do a simple and straightforward, but as with all such operations, you should make sure you have an unaltered copy of the Assembler stored safely on a disc before you begin. You will need to be careful that you alter the correct locations and put in the correct values: the Assembler will be unlikely to work correctly, and probably will not work at all, if you make a mistake.

25.20.1 The patch area

In order to patch the Assembler, you will first need to locate the patch table. This is an area of memory that either contains the actual locations you alter, or contains the addresses of the locations in other parts of the ROM.

The patch area is not held at a fixed absolute address, but will always follow the standard sideways ROM header. To find it, look first at byte $8007 in the ROM (in this section, all addresses are given from the base of $8000, which is the memory address the Assembler runs at. To get the offset within a file for the bytes concerned, simply subtract $8000 from all values). Add the value of byte $8007 to $8000, to obtain the address of the copyright string (in version 1.60 it reads ”(C)1987 Alan Phillips”).

At the end of the copyright string you will find a byte containing the value of zero; the patch area starts at the byte immediately following this one.

Athough the absolute position of the patch area is not fixed between releases, the format of it is, so that the following sections apply to 1.60 and any future releases.

25.20.2 The default TAB position list

Bytes 0 and 1 of the patch table are the address of the default TAB table used by the Assembler when listing source lines. (Note that, as usual, the address is stored with the least significant byte first).

The address in the patch table takes you to the area of the ROM where the default TAB positions are held. To change them, you should alter the first byte of the table to contain the number of TAB positions you wish to define (the number you use must be between 1 and 14). Following this you should place the TAB positions themselves; they must be in ascending numeric order.

For example, to define 5 default TAB positions at columns 8, 17, 25, 32 and 40, you would change the bytes in the TAB table to be 5 8 17 25 32 40

25.20.3 Source character translation

Byte 2 of the patch area controls how the Assembler treats characters in the source file with bit 7 set. Normally the value is zero, instructing the Assembler to ignore such bytes. Changing it to some other value makes the Assembler replace such bytes with the value you supply.

Thus, for example, you could change the byte in the patch table to the value $40 (the code for a space character) to replace source characters with bit 7 set with a space).

It is up to you to make sure that the value you use is sensible: you should set the byte either to a space or to a TAB character.

25.20.4 Real time clock handling

Byte 3 of the patch area controls how the Assembler uses any real-time clock you may have fitted. By default, the value is $FF, and the Assembler will attempt to read a real-time clock only on a Master 128. You may, though, have fitted a third-party clock to another model: setting the byte value to 0 will make the Assembler try to read the time on all models.

Note, though, that your real-time clock must respond to exactly the same *FX calls as used on the Master 128: the Assembler has no means of knowing how to read the time from clocks that are operated in other ways.

25.20.5 Character constant and string options

Byte 4 of the patch area controls how the " " character is treated in character constants and strings. By default the value of the byte is zero, and the character has no special effect. Changing the value to $FF causes the " " character to take on special meaning: it will now cause the following character to be planted as a control character with bit 7 set. Naturally, you would need to write " " in a character constant to obtain the " " character itself if you turn this option on.

25.20.6 Page throws at the end of a listing

Byte 5 of the patch area controls whether the Assembler outputs Form Feed characters to a printer or listing file at the end of a listing. By default, the value of the byte is $FF: this makes the Assembler output a Form Feed both before and after the lines of statistics that tell you the number of errors, etc. Changing the byte to 0 suppresses the Form Feeds, so the statistics immediately follow the symbol table listing, with no throw to head-of-form following them.

Page intentionally blank

The BBC Microcomputer

Table of contents

BBC Model B

Table of contents

26 Memory Map **565**

Memory Map 26

Page	BBC Model B	Notes
0000	Current Language Zero Page	00-8F is reserved for the current language but BBC Basic reserves 70-8F for user programs. If a program/game does not use Basic it can use the entire block
0070	User workspace in BASIC	
0090	Operating System Zero Page	
0100	Processor Stack	
0200	Operating System Workspace	
0300		
0400	Current Language Workspace	1K reserved for the current language but if a program/game does not use the language it can use this space.
0500		
0600		
0700		
0800	Sound and Printer Buffer	
0900	RS423, Speech, Tape output buffer	
0A00	RS423 and Tape input buffer	
0B00	Function Keys	
0C00	User defined graphics	
0D00	NMI routines (Disk/Econet)	
0D9F	Expander vector set	
0DF0	Paged ROM workspace locations	
0E00	USER workspace	USER workspace can start later if a Paged ROM needs workspace. *e.g.* DFS can push the start to 1900 and ADFS to 1A00.
1000		
2000		
3000	Mode 0, 1, 2	USER workspace ends at the start of the current display mode
4000	Mode 3	
5800	Mode 4, 5	
6000	Mode 6	
7B00	Mode 7	
8000	16K Paged ROM	
C000	OS ROM	
FC00	FRED 1MHz Bus	Memory mapped I/O devices. These are generally the same across systems but there are some differences, especially with the Electron.
FD00	JIM 1MHz Bus	
FE00	SHIELA I/O Memory	
FF00	OS ROM Vectors	

Figure 26.1: BBC B Memory Map

Page intentionally blank

BBC Master 128

Page intentionally blank

Memory Map 27

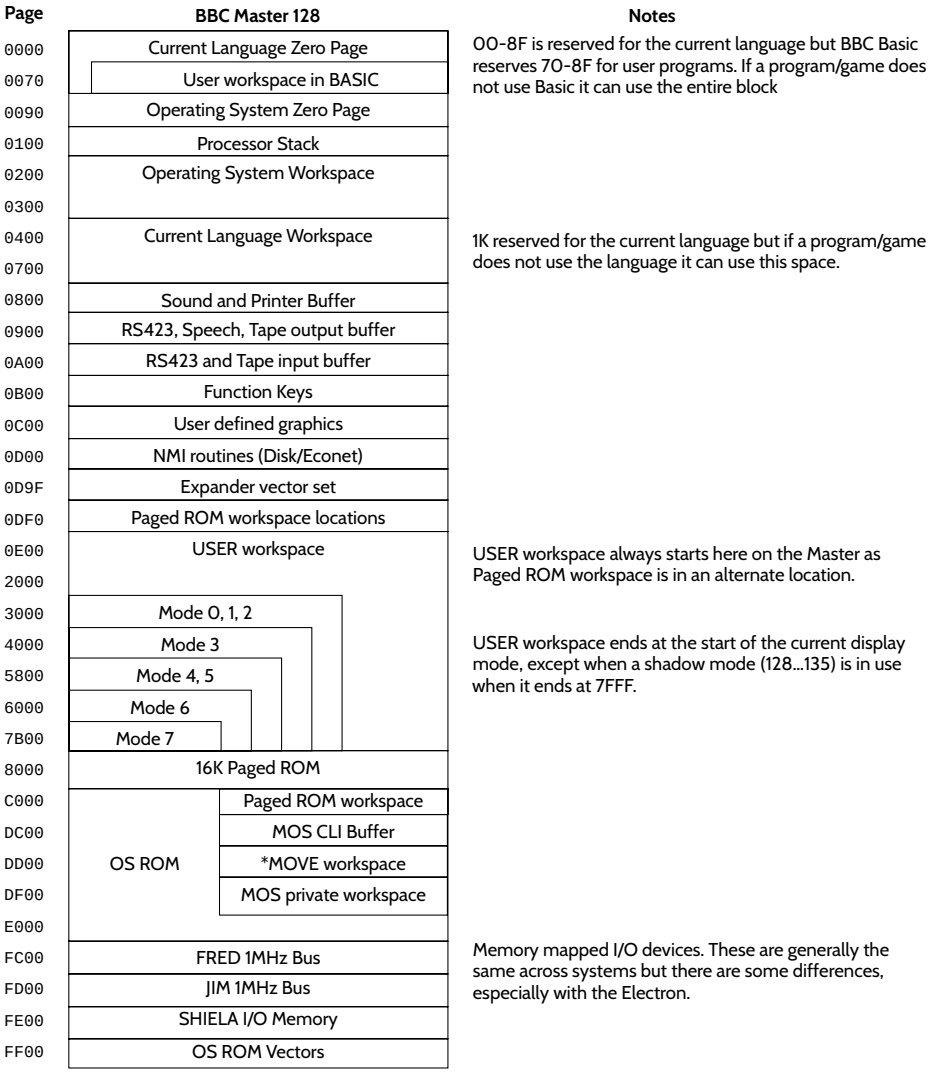


Figure 27.1: BBC Master 128 Memory Map

Page intentionally blank

28 Configuration

After the Master series have had either their batteries or EEPROM replaced it will normally be necessary to perform some sort of re-configuration. This chapter[7] describes all the configuration options available and what the default settings are when the computer is supplied by Acorn.

[7] *Support Group Application Note No. 203*, (Feb. 27, 1992)

This only applies to the BBC Master 128, 512, ET and Compact models.

On the master series, the configuration is managed with the ***CONFIGURE** command[1]. This takes one or two parameters, the setting to set and an optional value for that setting.

1: Like other commands it can be shortened as ***CONF.** Note the . at the end is required when using the shortened version.

The ***CONFIGURE** commands must always be followed by a CTRL BREAK to initialise the settings in the machine.

It is possible to use Edit supplied with the BBC Master to create a standard file which contains all the ***CONFIGURE** options which could be used to re-configure the machine in one go by using the ***EXEC** command.

The Master Compact EEPROM has a limited life of 10,000 write operations. The chip is easily replaced when this limit is reached.

If ROMs appear in the ROM table as being unplugged it is possible to re-insert then using the command ***INSERT** followed by the ROM Socket number eg. ***INSERT 13**

28.1 Default configuration

Table Table 28.1 on page 572 shows the default configuration for a BBC Master.

The Master Compact uses an EEPROM rather than battery backed memory.

This is handy as, if the battery fails or is replaced then the configuration is wiped requiring it to be re-entered.

Table 28.1: Default configuration for the BBC Master

Configuration	Default	Description
BAUD	4	1200
BOOT/NOBOOT	NOBOOT	Do not boot on power up
CAPS/NOCAPS	CAPS	Caps lock on
DATA	4	8N2
DELAY	30	0.30s
DIR/NODIR	NODIR	Do not initialise directory
FDRIVE	0	6ms, ADFS precompensation enabled
FILE	13	ADFS
HARD/FLOPPY	FLOPPY	Default to floppy disk
IGNORE	0	No ignore character
INTUBE/EXTUBE	INTUBE	Internal Tube Interface
LANG	12	BASIC
Printer	0	Sink
QUIET/LOUD	QUIET	
MODE	7	Teletext Mode 7
REPEAT	8	0.08s
SCROLL/NOSCROLL	SCROLL	

28.2 ANFS

The following options only apply to systems fitted with the ANFS (Advanced Network Filing System) ROM.

28.2.1 *CONFIGURE FS <x.xxx> or <Name>

Used to selected the default fileserver for logging onto.

Options:

FS Number
FS Name

Examples:

***CONFIGURE FS 0,254**

***CONFIGURE FS TecSupport**

Default:

0,254

28.2.2 *CONFIGURE PS <x.xxx> or <Name>

Used to selected the default network printer server.

Options:

FS Number
FS Name

Examples:

***CONFIGURE FS 0,235 *CONFIGURE FS Laser**

Default:

0,235

28.2.3 *CONFIGURE SPACE/NOSPACE

On the BBC Master Series pages 0x0B00 and 0x0C00 are allocated as workspace to be used by the Econet filing system.

On previous models these areas were used a buffers. Some existing software may try to use these areas and conflict with the operating system use of the space.

The Econet system can be configured to claim other areas thus leaving pages 0x0B00 and 0x0C00 free for use with any application programs which are found to make inappropriate use of these areas.

Options:

SPACE Pages claimed for transient commands. Effect on PAGE, OSHWM rises by 0x200.

NOSPACE PAGEs are not claimed and therefore the OSHWM is not increased.

28.3 *CONFIGURE Baud <0...8>

Sets the speed of the RS423 serial interface.

Options:

Setting	Baud rate
0	9600
1	75
2	150
3	300
4	1200
5	2400
6	4800
7	9600
8	12000

Default:

4 1200

28.4 *CONFIGURE Boot / NoBoot

Sets the boot action when the machine is switched on. If set to BOOT then the machine will attempt to run the !BOOT file on a disk on switchon or CTRL Break.

Options

BOOT On startup/CTRL Break, attempt to run !BOOT file on disk 0

NOBOOT Do nothing on boot

Default

NOBOOT

28.5 *CONFIGURE Caps / NoCaps

Sets the keyboard caps lock

Default:

Caps

28.6 *CONFIGURE Data <0...7>

Sets the data format used by the RS423 serial interface.

Options:

Value	Abbrev	Word Len	Parity	Stop Bits
0	7E2	7	even	2
1	7O2	7	odd	2
2	7E1	7	even	1
3	7O1	7	odd	1
4	8N2	8	none	2
5	8N1	8	none	1
6	8E1	8	even	1
7	8O1	8	odd	1

Default:

4 8N2

28.7 *CONFIGURE Delay <0...255>

Sets the keyboard auto-repeat delay

Default:

30

28.8 *CONFIGURE Dir/NoDir

Initialise ADFS with a directory

Options:

DIR Causes ADFS to initialise with a directory selected (either the root or, if ADFS has been since power on, the previously selected directory).

NODIR Do not select a directory

Default:

NODIR

28.9 *CONFIGURE FDrive <0...7>

For computers fitted with either WD1770 or WD1772 disc controllers. FDrive defines the step time and, for ADFS, use of precompensation.

Options:

Value	track step time		precompensation
	WD1770	WD1772	ADFS only
0 or 4	6ms	6ms	yes
1 or 5	6ms	6ms	no
2 or 6	30ms	3ms	yes
3 or 7	30ms	3ms	no

Default:

0

28.10 *CONFIGURE File <0...15>

Select the default file system by it's ROM socket.

e.g. ***CONFIGURE FILE 9** would select DFS as the new default filesystem.

Default:

13

28.11 *CONFIGURE Hard/Floppy

For systems that have both floppy disc(s) and a Winchester disc connected:

***CONFIGURE FLOPPY** causes ADFS to initialise with drive 4 (floppy discs) selected.

***CONFIGURE HARD** causes ADFS to initialise with drive 0 (Winchester) selected.

Default:

FLOPPY

28.12 *CONFIGURE Ignore <0...255>

Defines the 'printer ignore character' (ie the character which is not to be directed to the printer), by means of its ASCII code.

If the character code is omitted, all characters are sent to the printer when enabled.

Default:

0

28.13 *CONFIGURE Lang <0...15>

Select the default language by it's ROM socket.

Default:

12

28.14 *CONFIGURE Mode <0...7, 128...135>

Select the default screen mode

Default:

7

28.15 *CONFIGURE Print <0...4>

Sets the printer port

Options:

Default:

0 Sink

Value	Port
0	Sink *(No output printed)*
1	Centronics Parallel Port
2	RS423 Serial Port
3	User printer driver
4	Econet network printer

28.16 *CONFIGURE Quiet/Loud

Sets the Bell (CTRL G) Volume

Default:

QUIET

28.17 *CONFIGURE Repeat <0...255>

Sets the keyboard repeat rate in 1/100 second

Default:

8

28.18 *CONFIGURE Scroll / NoScroll

SCROLL enables the scroll protect option (ie prevents hard screen scrolling if a character is placed in the last character position of the bottom screen line, depending on the current mode.

Default:

SCROLL

28.19 *CONFIGURE Tube / NoTube

If TUBE is enabled then, via the INTUBE or EXTUBE option, check for its existence and enable it

Default:

NOTUBE

28.20 *CONFIGURE View Setup <XXX>

The VIEW option only applies to BBC Master Compact and to BBC Master 128's that are fitted with the optional System ROM (3.50).

Define how View should start up.

Options:

F Format mode is on when View starts (an F will appear at the top left of the screen)
J Justify mode is on when View starts (a J will appear at the top left of the screen)
I Insert mode is on when View starts (an I will appear at the top left of the screen)

Default:

FJ

Page intentionally blank

Acorn Electron

Page intentionally blank

MOS
Machine Operating System

Table of contents

29 Keyboards **587**
29.1 Background processing . . . 587
29.2 Break key . . . 588
29.3 Reading keys . . . 589
29.4 Keys and layouts . . . 590

30 Operating System Calls **595**
30.1 OSWRCH - Write character . . . 596
30.2 OSNEWL - Write newline . . . 596
30.3 OSASCI - Write character, call OSNEWL if A=0x0D . . . 597
30.4 OSRDCH - Read character . . . 598
30.5 OSBGET Read byte from open file . . . 599
30.6 OSBPUT Write byte to open file . . . 599
30.7 OSCLI - Pass string to CLI . . . 600
30.8 OSEVEN - Generate an event . . . 602
30.9 OSRDSC - Read byte from Screen or paged ROM . . . 602
30.10 OSBYTE - OS functions by A . . . 603
30.11 OSWORD - OS functions with a parameter block . . . 751

31 Operating System Commands **777**
31.1 *CODE x,y . . . 777
31.2 *FX a,x,y Perform OSBYTE call . . . 777
31.3 *LINE text . . . 778
31.4 *MOTOR n . . . 778
31.5 *TAPE n . . . 778
31.6 *TV s,i . . . 779
31.7 Media . . . 780

29 Keyboards

The keyboards on the BBC Model B, BBC Master 128, BBC Master Compact and Acorn Electron all have different keyboard matricies which the developer would have to take account for if they wanted to read the keyboard directly.

To eliviate this, a system of internal[1] key codes was used. This was done to ensure that software written originally for the BBC Model B would still work with the later machines even though they had either additional keys like the BBC Master 128 or absent keys like on the more compact Acorn Electron.

1: These codes actually relate to the original keyboard matrix

Then instead of reading the keyboard directly, the developer could use one of the standard API's.

29.1 Background processing

In MOS, the keyboard is scanned as part of the systems 10*ms* interrupt cycle[4]. When a key press is detected the key press is decoded and the internal key code of the key press is stored in zero-page. As the routine decoding the keyboard operates a 2-key Rollover then up to two keys are stored.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.218

In zero-page, if a key is pressed then it's stored with bit 7 set. If no key is pressed then 0 is stored.

Table 29.1 shows the memory locations used[8].

[8] *Adv. User Guide, Electron*, Dickens (Adder Publishing, Cambridge June 2008) P.39

Once the keyboard scan has completed, the detected key is then decoded into ASCII and placed into the keyboard buffer. This also takes note on when the key was pressed so that, after the appropriate intervals, the key will then repeat until the physical key is released.

Table 29.1: Last Key Pressed

Keys	0xEC	0xED
None pressed	`0x00`	`0x00`
One pressed	`First key`	`0x00`
Two pressed	`Second key`	`First key`
Second released	`0x00`	`First key`

29.2 Break key

On the BBC Master 128, there was a small cam next to the Break key which could be turned with a small screwdriver, so that the key could not be pressed accidentally.

On all the machines, the Break key was not part of the keyboard matrix. This means that this key cannot be read, so it is not possible for software to detect if it is pressed or not.

This is because it was connected directly to the processor's reset line, so when it was pressed it reset the machine. Unlike most machines of the era, this restart did not necessarily wipe the machine – although that was possible via an OSBYTE call.

When Break was pressed, if the shift key was also held indicated that the user wanted to boot from a disc. This usually involved reading the file `!BOOT` and executing the commands in that file.

If the Control key was pressed indicated that the user wanted to reset the computer completely. This wasn't exactly the case, as the machine usually kept its memory intact[2], but reset the machine to a state closer to an inital power on state than a simple break.

Finally, if a letter on the keyboard was pressed, then a specific filesystem was selected.

e.g. D+Break would start Acorn DFS, whilst A+Break Acorn ADFS. T and R were common to select the Tape and ROM filesystems.

2: If *FX200,3 had been issued then memory is wiped on a break. See **OSBYTE `0x7C`** Subsection 30.10.48 on page 636

29.3 Reading keys

In most instances applications use the **OSRDCH**[3] routine to read from the current input stream. This allows for characters to be read from either the keyboard, serial or a file.

3: **OSRDCH** Section 30.4 on page 598

Alternatively **OSBYTE** `0x81`[4] or BBC Basics **INKEY** function can be used. This has the benefit that you can check for a key pressed within a time limit or look for a specific key press – handy in games.

4: **OSBYTE** `0x81` ??

Both of these calls use either ASCII, inkey or internal codes depending on what action is being performed.

Table D.11 lists the keys on all three keyboard layouts, along with the ASCII, INKEY and internal key codes.

Table D.11: P.988

29.3.1 ASCII

The first is ASCII, or more specifically the BBC Microcomputers VDU character set which is used by calls like **OSRDCH**[5] when reading from the input stream, itself usually the keyboard.

5: **OSRDCH** Section 30.4 on page 598

The OSBYTE/INKEY method uses ASCII when reading the keyboard with a timeout.

29.3.2 INKEY

The INKEY code is used by the BBC Basic **INKEY** function the OSBYTE `0x81` call[6] when passed a negative value.

6: call ??

When **INKEY** or the related OSBYTE call is passed a negative value then that number is used to test to see if a specific key is pressed.

This negative value is actually the internal key code exclusive-ORed with `0xFF`.

29.3.3 Internal Key Code

Lastly there's the internal key codes. These codes relate to the position of the keys within the physical keyboard matrix of the original BBC Model B keyboard.

29.4 Keys and layouts

All the keyboards used real keys which, at least for the BBC Microcomputer made a distinctive sound when pressed. Even the Acorn Electron had real keys which distinguished it from the other machines available at the time at that price point.

29.4.1 BBC Model B keyboard

Figure 29.2 on page 593 shows the keyboard for the BBC Model B and BBC Model B+. It consists of 74 keys as well as a set of eight dip switches which were included as part of the matrix. These show up in the table as keys, and are used to determine how the machine should start up.

These switches were removed in later machines. The BBC Master 128 used Battery Backed RAM to store this configuration whilst the BBC Master Compact used EEPROM.

29.4.2 BBC Master 128 and BBC Master Compact keyboards

Figure 29.3 on page 594 shows the keyboard for the BBC Master 128 and BBC Master Compact. This had 93 keys consisting of the 74 from the original BBC Model B as well as an additional 19 in the form of as numeric keypad.

The keys on the keypad were independent to those on the main keyboard, so they came with their own internal key codes and could be inspected separately.

In normal use, the shift key has no effect on the keypad. This could be changed with **OSBYTE `0xFE`**[7].

7: **OSBYTE `0xFE`**
Subsection 30.10.199

29.4.3 Acorn Electron keyboard

Figure 29.1 on page 591 shows the keyboard for the Acorn Electron. It comprised of 56 keys with the removal of the separate function and tab keys as well as the shift lock.

Instead, the caps lock key became multifunctional. It either toggled caps lock or acted like a shift key so that the second key pressed would be a function key.

This meant that the original Function keys 0 ... 9 were now accessible via the normal digits. In addition, pressing Func and a letter automatically typed common basic instructions similar to how the spectrum worked – except here it was optional, you could still type the command as normal.



Figure 29.1: Acorn Electron Keyboard

Erratta

With the Acorn Electrons keyboard being smaller than the others, it means that some key codes exist for keys which are not present on the Electron. Even with recent modern revisions of various user guides show the codes for the BBC Model B when mentioning the Electron.

e.g. In the New Advanced User Guide[4] it states the Electron keys are identical to the BBC Model B except for the option switches.

In the Electron Advanced User Guide[8] those switches, which are used on the original BBC Model B to define how the machine starts, are actually listed along with non-existent keys like Scroll Lock.

In Table D.11 I have hopefully solved this by listing which codes should work with which keyboards by using the same data used for the keyboard layouts in this chapter.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.219

[8] *Adv. User Guide, Electron*, Dickens (Adder Publishing, Cambridge June 2008) P.40

Table D.11: P.988



Figure 29.2: BBC B Keyboard



Figure 29.3: BBC Master 128 Keyboard

30 Operating System Calls

The core MOS OS calls are identical on the BBC Microcomputer, BBC Master 128 and Acorn Electron so are shown in this chapter. Differences between the three systems will be shown against the individual entry.

Table D.1 lists the main entry points along with their indirection vectors.

Table D.1: P.965

30.1 OSWRCH - Write character

Function OSWRCH

Address FFEE

Vector 020E

Description Write VDU character in A to screen

On entry

A VDU code of character

On exit

A, *X*, *Y* Preserved

C, *N*, *V*, *Z* Undefined

This routine writes the character in the accumulator to the currently selected output stream.

The A, X and Y registers are unmodified on it's return.

30.2 OSNEWL - Write newline

Function OSNEWL

Address FFE7

Description Write LF then CR to screen

On exit

A `0x0D`

X, *Y* Preserved

C, *N*, *V*, *Z* Undefined

This routine writes the sequence `0x0A`, `0x0D` to the current output stream, causing a new line to be started.

The A register will be set to `0x0D` on its exit.

30.3 OSASCI - Write character, call OSNEWL if A=0x0D

Function OSASCI

Address FFE3

Description Write char in A to screen. If the char is CR (`0x0D`) then write a LF (`0x0A`).

On entry

A VDU code of character

On exit

A, *X*, *Y* Preserved

C, *N*, *V*, *Z* Undefined

This routine will write the character in A to the currently selected output stream just like **OSWRCH** . However, if A=`0x0D` then this will write a line feed (`0x0A`) first, causing a new line to be started.

Listing 30.1 on page 597 shows the actual source for the **OSASCI**, **OSNEWL** and **OSWRCH** MOS calls.

Listing 30.1: OSASCI source

```
1         ORG 0xFFE3
2
3  .OSASCI CMP #0x0D        ; If not CR then
4         BNE OSWRCH       ; jump to OSWRCH
5
6  .OSNEWL LDA #0x0A        ; Write a Line Feed
7         JSR OSWRCH
8         LDA #0x0D        ; Restore A to be CR
      then
9                          ; fall through to OSWRCH
10
11 .OSWRCH JMP (0x020E)     ; Jump to WRCHV vector
```

30.4 OSRDCH - Read character

Function OSRDCH

Address FFE0

Vector 0210

Description Read a character from the current input stream into A

On exit C=0

A ASCII character read

X, *Y* Preserved

N, *V*, *Z* Undefined

On exit C=1

A Error number

X, *Y* Preserved

N, *V*, *Z* Undefined

If an escape condition occurs then A=`0x1B` and must be acknowledged with **OSBYTE `0x7E`**[1]. This routine reads a character from the currently selected input stream and returns it in the accumulator.

If an error should occur acknowledgement of the error condition should be made using OSBYTE 0x7E **??**.

1: OSBYTE `0x7E`
Subsection 30.10.50

30.5 OSBGET
Read byte from open file

Function	OSBGET
Address	FFD7
Vector	0216

30.6 OSBPUT
Write byte to open file

Function	OSBPUT
Address	FFD4
Vector	0218

When used on a Second Processor, then **OSBYTE `0x9D`**[2] is a potential and faster alternative to this call.

2: OSBYTE `0x9D`
Subsection 30.10.87 on page 665

30.7 OSCLI - Pass string to CLI

Function OSCLI

Address FFF7

Vector 0208

Description Pass a line of text to the command line interpreter. The line of text should be terminated by a carriage return `0x0D`.

On entry

XY The address of the line of text

On exit

A, *X*, *Y* Undefined

C, *N*, *V*, *Z* Undefined

OSCLI provides the machine code user with a convenient method of performing any of the * commands that the system provides from Basic.

The command required is placed in a string as normal text terminated by a CR character and this call is made with the address in X and Y.

If the string passed to the CLI is not terminated by a carriage return within 255 bytes this routine has undefined effects.

Note these have * in them but that is not necessary when passing the command to **OSCLI**

The OS provides a default set of commands listed in Table 30.1 on page 601.

These commands may be abbreviated by taking the first few letters and terminating with a '.' character. Parameters may be passed in the text following the command.

Any unrecognized commands are first offered to paged ROM's and if not claimed and are then offered to the currently selected filing system via the filing system control vector.

Table 30.1: OS Provided OSCLI commands

Command	Action	Available on BBC	Master	Electron
*.	treated as *CAT command	X	X	X
*/	treated as *RUN command	X	X	X
*BASIC	Select BASIC as current language	X	X	X
*CAT	issue catalogue request to filing system	X	X	X
*CODE	passed to user vector	X	X	X
*EXEC	select text file as input stream	X	X	X
*FX	issue OSBYTE call, no returns possible	X	X	X
*HELP	issue page ROM service call 9	X	X	X
*KEY	take rest of line as text for soft key	X	X	X
*LINE	passed to user vector	X	X	X
*LOAD	issue load request to filing system	X	X	X
*MOTOR	open/close cassette motor relay	X	X	X
*OPT	issue option request to filing system	X	X	X
*ROM	select ROM filing system	X	X	X
*SAVE	issue save request to filing system	X	X	X
*SPOOL	include text file in output stream	X	X	X
*TAPE	select tape filing system	X	X	X
*TV	TV configuration	X	X	

30.8 OSEVEN - Generate an event

Function OSEVEN
Address FFBF

The user event may be generated using this routine. Software replacing OS routines should generate the appropriate events by making this call.

30.9 OSRDSC - Read byte from Screen or paged ROM

Function OSRDSC
Address FFB9

This call is not available from second processors.

This call returns a byte read from a paged ROM. The ROM number is in Y. The address to read is in 0xF6 and 0xF7

This routine was included for the implementation of ROM filing system software in paged ROM and is not recommended for general use.

In the original Advanced User Guide[5] this was called `OSRDRM` and was used to read from Paged Rom. However, in the New Advannced User Guide[4] this routine had been renamed to `OSRDSC`.

[5] *Adv. User Guide, BBC Micro*, Bray (Cambridge Microcomputer Center Nov. 1983) P.106

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.113

30.10 OSBYTE - OS functions by A

Function	OSBYTE
Address	FFF4
Vector	020A

OSBYTE calls are a powerful and flexible way of invoking many of the available operating system facilities.

They are used for functionality which only requires a simple one or two byte parameter to the function and similarly for the return.

More complex calls are implemented via the **OSWORD**[3] call which takes a variable length parameter block.

3: **OSWORD** Section 30.11

The remainder of this chapter describes each documented **OSBYTE**.

Table D.2 lists the documented **OSBYTE** calls below `0xA6` and which platforms support them.

Table D.2: P.967

For calls `0x`$A6 \ldots FF$ which provide access to the MOS variables are listed in Table D.8 listing the variables themselves. These two can vary between individual platforms.

Table D.8: P.978

30.10.1 Parameters

OSBYTE uses the Accumulator (A register) to define the action. This means that up to 256 different calls can be made.

Any parameters are passed to the call via the X and Y index registers.

Calls are grouped into three types defined by the value of the accumulator.

$A \leq 0x7F$	Only X is used for the parameter
$A \geq 0x80$	X and Y pass values to the used for the parameter
$A \geq 0xA6$	Read and Write operation using both X and Y registers

Calls where $A \geq$ 0xA6 MOS Variables

Calls where $A \geq$ 0xA6 provide both read and write access to the MOS variables using both the X and Y registers.

The way those calls operate is defined by the following formula:

$$new\ value = (old\ value\ \wedge\ Y)\quad X$$

To read the variable then X needs to be set to 0x00 whilst Y to 0xFF. Setting the variable, replacing it completely then Y needs to be set to 0x00 and X to the value to set.

Changing individual bits

You can update only specific bits of a value, leaving the rest unchanged.

To do this you need to set Y to mask out the bits you want to change, and X to the bits you want to set. Table 30.2 shows the common masks for each bit for Y and the value of X to set a bit. For clearing a bit then X is 0 every time.

e.g. to set bit 7 you need to set X to 0x80 and Y to 0x7F.

Table 30.2: Bit masks for OSBYTE calls $A \geq$ 0xA6

Bit	X	Y Mask
0	0x01	0xFE
1	0x02	0xFD
2	0x04	0xFB
3	0x08	0xF7
4	0x10	0xEF
5	0x20	0xDF
6	0x40	0xBF
7	0x80	0x7F

OSBYTE FF[4] is an example where bit 3 determines how !BOOT operates after a soft break. In that instance $Y = 0xF7$ and X is either 8 to set or 0 to clear the option.

In *FX format this would look like:

```
*FX255,8,247    !BOOT after SHIFT+Break
*FX255,0,247    !BOOT after Break
```

Toggle individual bits

Toggling bits is virtually the same process as setting or clearing individual bits. Here, X is set to the bits to toggle and Y is set to 0xFF. So to toggle bit 3 as above, then $X =$ 0x08 as if you were setting it but the mask in Y is always 0xFF.

4: **OSBYTE FF** Subsection 30.10.4 on page 608

Returning values

Many **OSBYTE** calls can return values in either the X or both X and Y registers:

For calls where $A \leq$ `0x7F` then only the X register contains any result. The Y register and flags are. unless otherwise documented, usually in an unknown state following the call.

For calls where $A \geq$ `0x80` then both the X and Y registers return a value, with some also utilising the Carry flag.

Calls with $A \geq$ `0xA6`

The calls which read and write the MOS variables all return the original value in the X register. This value is that before any update was made to it when a write was performed.

The Y register is set to the value of the next byte in memory. In many instances this is the next variable, but for those variables which are 16-bit values like addresses, this is the high most significant byte of that variable. This allows for those variables to be read in a single **OSBYTE** call.

Updates to those multiple byte variables are only performed on the low least-significant byte. To update the high byte a second **OSBYTE** call is needed with A incremented byt one.

No return

It is possible for a **OSBYTE** call never to return.

One example is with **OSBYTE `0x00`**[5] when $X = 0$ will cause a software break to be invoked so the result is sent as an error to the currently active language ROM. For that call, any other value for X returns to the caller.

5: OSBYTE `0x00`
Subsection 30.10.4

Another example is **OSBYTE `0x8E`**[6] which will never return ever as that call enters the numbered language ROM so that language takes the machine over.

6: **OSBYTE `0x8E`** Subsection 30.10.68

30.10.2 Calling OSBYTE

There are two ways to call **OSBYTE**:

Machine Code

Function	OSBYTE
Address	FFF4
Vector	020A

From machine code, **OSBYTE** can be called by calling `0xFFF4` with the call number in the Accumulator (A register) and the parameters in the X and Y index registers.

The entry point at `0xFFF4` will then use the indirection vector at `0x020A` to send the call to the actual routine. This vector allows for user code to intercept **OSBYTE** calls before they get to the operating system.

Any unsupported **OSBYTE** calls are sent to Paged ROMs as a service call.

*FX

The command line interpreter **OSCLI**[7] will perform an **OSBYTE** call in response to a ***FX**[8] command, enabling the user to issue **OSBYTE** calls from the command line.

7: **OSCLI** Section 30.7

8: ***FX** Section 31.2

Most language ROMs generally pass commands beginning with `*` to **OSCLI**.

It should be noted however that no results are returned by a ***FX** call and so it is inappropriate to use certain **OSBYTE** calls that return values in this way.

30.10.3 Errata

The content of this section has been taken from multiple sources[4,8,9] but primarily from the original User Guides by Acorn, the Advanced User Guides for the BBC Microcomputer and Acorn Electron from the era as well as the modern reprints, updates and some websites.

However, there are some inconsistencies where some sources state one thing for a call whilst a more recent one either conflicts with it or only mentions a newer version of the call ignoring the older version.

When I've spotted this I include an Errata section with the call describing what differs, what is obviously wrong along with the sources that have the error.

An example of this is the buffer id's used by **OSBYTE `0x15`**[9] and **OSBYTE `0x80`**[10]. Many sources, including modern ones, state they use the same ID's but they do not so this is mentioned against both calls.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022)
[8] *Adv. User Guide, Electron*, Dickens (Adder Publishing, Cambridge June 2008)
[9] *Master Series Ref. Manual Part 1*, (Jan. 1986)

9: **OSBYTE `0x15`** P.622
10: **OSBYTE `0x80`** P.638

30.10.4 OSByte 0x00 (0) Get OS Version

Description Execute BRK and print OS version.

On entry

A = `0x00`

$X = 0$

On exit This call never returns as control is handed to the break handler within the current language.

Description Return with OS version in X

On entry

A = `0x00`

$X \neq 0$

On exit

Carry Undefined

A, Y Preserved

X See Table 30.3

Table 30.3: P.608

11: OSBYTE `0x81` Subsection 30.10.55

Table D.10: P.983

OSBYTE `0x81`[11] with $X = 0, Y =$ `0xFF` also returns a slightly different list of versions and machines in Table D.10

Table 30.3: OS versions

X	Version	Machine Type
0	1.00	Early BBC Model B
0	E1.00	Acorn Electron
1	1.20	BBC Model B
1	A1.00	BBC Model B American OS
2	2.00	BBC Model B+
3	3.20	BBC Master 128
3	3.50	BBC Master 128
4	4.00	BBC Master Econet Terminal
5	5.00	BBC Master Compact

Errata

The New Advanced User Guide[4] shows differing values for OS versions.

In Table 30.3 the version for the BBC Master Compact is 5.0. However, in Table D.10 for **OSBYTE `0x81`**, the version is 5.10.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.428
Table D.10: P.983

30.10.5 OSByte 0x01 (1) Write user flag

Description Set the user flag

On entry

A = `0x01`

X New value of user flag

On exit

X Old value of user flag

The user flag is not used by any version of MOS and is available for use by the end user.

OSBYTE `0xF1`[12] is an alternate **OSBYTE** call which also accesses this flag.

12: OSBYTE `0xF1`
Subsection 30.10.185

30.10.6 OSByte 0x02 (2) Specify input stream

Description Select the input stream.

On entry

A = `0x02`

X See Table 30.4

Table 30.4: P.610

The default is $X = 0$ and the RS-423 port is disabled for input.

If data needs to be received whilst the keyboard is in use then you need to use $X = 2$ so the port is enabled and data enters the RS-423 input buffer until it can be read.

Example

An example of this is in a Terminal Emulator. Here you would select $X = 2$ so that data from the remote end of the RS-423 connection is received. When data is received it generates an input event[4] so it can be displayed in real time.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.146

Whilst this is happening, the main part of the emulator could be waiting for input from the keyboard. When a key is pressed, it too generates an input event which can then cause the key to be sent to the remote server.

Table 30.4: Input Streams

X	Keyboard	RS-423
`0`	selected	sink input
`1`	buffer input	selected
`2`	selected	buffer input

30.10.7 OSByte 0x03 (3) Specify output stream

Description Select the output stream.

On entry

A = `0x03`

X See Table 30.5

Table 30.5: P.611

On the Acorn Electron, if either printer or RS-423 output is selected it is passed to paged rom service calls. If no ROM claims the call then the output is sunk.

Bit 3 should not be used as can conflict with Econet.



X bit	Output if set
`0`	Enable RS-423 driver
`1`	Disable VDU driver
`2`	Disable printer driver
`3`	Enable printer independent of VDU 1 or 2
`4`	Disable spooled output
`6`	Disable printer unless preceded by VDU 1

Table 30.5: Output Streams

30.10.8 OSByte 0x04 (4) Enable/disable cursor editing

Description Enable or disable cursor editing.

On entry

A = `0x04`

X See Table 30.6

On exit

X Old editing key status

X = 3 only applies to the BBC Master Compact where the cursor keys emulate the joystick and the copy key is the fire button.

Table 30.6: P.612

Table 30.6: Cursor editing mode

X	Status	Notes
`0`	enabled	Default
`1`	disabled	Keys generate ASCII codes 135 ... 139
`2`	disabled	Keys act as F_{11} ... F_{15}
`3`	joystick	BBC Master Compact only, copy=fire

30.10.9 OSByte 0x05 (5) Select printer destination

Description Select printer destination

On entry

A = `0x05`

X See Table 30.7

Table 30.7: P.613

On exit

X Old editing key status

For the Acorn Electron it is 0 unless the Plus 1 is present

The default is 1. A soft break does not reset this setting.

The RS-423 printer will sink if it is also enabled for normal output via **OSBYTE `0x03`**[13].

13: OSBYTE `0x03` Subsection 30.10.7

Table 30.7: Printer destination

X	Destination
`0`	Sink printer output
`1`	Parallel output
`2`	RS-423 output, sink if RS-423 enabled
`3`	User printer routine
`4`	Network printer
`5…255`	User printer routine

30.10.10 OSByte 0x06 (6) Write printer ignore character

Description Select printer ignore character

On entry

A = `0x06`

X Character to ignore

On exit

X Old ignored character

30.10.11 OSByte 0x07 (7) Set RS-423 receive rate

Description Select RS-423 receive baud rate

On entry

A = `0x07`

X See Table 30.8

Table 30.8: P.614

On exit Not on the Acorn Electron

X, Y Old serial ULA register contents

On exit Acorn Electron

X, Y Undefined

Table 30.8: RS-423 Baud rates

X	Baud Rate
0	9600
1	75
2	150
3	300
4	1200
5	2400
6	4800
7	9600
8	19200

30.10.12 OSByte 0x08 (8) Set RS-423 transmit rate

Description Select RS-423 transmit baud rate

On entry

A = `0x08`

X See Table 30.8

Table 30.8: P.614

On exit

X, Y Same as **OSBYTE `0x07`**

30.10.13 OSByte 0x09 (9) Set duration of first colour

Description Set the mark duration of flashing colours

On entry

A = `0x09`

X = 0 Set mark duration to infinity[14]

$X \neq 0$ Mark duration in $\frac{1}{50}$ of a second

On exit

X Old mark duration

The default mark duration is 25.

14: If space is also 0 then this forces the colours into the mark state

30.10.14 OSByte 0x0A (10) Write duration of second colour

Description Set the space duration of flashing colours

On entry

A = `0x0A`

X = 0 Set space duration to infinity[15]

$X \neq 0$ Space duration in $\frac{1}{50}$ of a second

On exit

X Old mark duration

The default space duration is 25.

15: If mark is also 0 then this forces the colours into the space state

30.10.15 OSByte 0x0B (11) Set keyboard auto-repeat delay

Description Select the time before the keyboard will repeat a key being pressed

On entry

A = `0x0B`

X = 0 Disable auto-repeat of keys

$X \neq 0$ Delay in centiseconds

On exit

X Old delay duration

The default keyboard repeat delay is $50cs$[16]

16: $50cs = 0.5s$

30.10.16 OSByte 0x0C (12) Set keyboard auto-repeat rate

Description Select the time between repeats once the keyboard starts repeating a key

On entry

A = `0x0C`

X = 0 Reset delay and period to default values

$X \neq 0$ Repeat interval in centiseconds

On exit

X Old repeat interval

The default keyboard repeat interval is $8cs$.

30.10.17 OSByte 0x0D (13) Disable event

Description Disable events processed by **OSEVEN**[17]

17: **OSEVEN** P.602

On entry

A = `0x0D`

X See Table 30.9

Table 30.9: P.617

On exit

$X = 0$ Event was previously disabled

$X > 0$ Event was previously enabled

X	Event Type
`0`	Output buffer became empty
`1`	Input buffer became full
`2`	Character inserted into input buffer
`3`	ADC conversion complete
`4`	Start of V-Sync
`5`	Interval timer crossing zero
`6`	ESCAPE condition detected
`7`	RS-423 error detected
`8`	Econet event
`9`	User event

Table 30.9: Event types for OSEVEN

30.10.18 OSByte 0x0E (14) Enable event

Description Enable events processed by **OSEVEN**[18]

18: **OSEVEN** P.602

On entry

A = `0x0E`

X See Table 30.9

Table 30.9: P.617

On exit

X Same as **OSBYTE `0x0D`**

30.10.19 OSByte 0x0F (15) Flush buffers

Description Flush either the current input buffer or all buffers

On entry

A = `0x0F`

X = 0 Flush all buffers

$X \neq 0$ Flush current input buffer

This call is a general purpose input buffer flush which can be applied to either all buffers or just the current input buffer.

To flush a specific buffer you need to use **OSBYTE `0x15`**[19]

19: **OSBYTE `0x15`** P.622

30.10.20 OSByte 0x10 (16) Select ADC channel

Description Select number of ADC channels to be sampled

On entry

A = `0x10`

X = 0 Sampling disabled

X = n Number of channels to be sampled

On exit

X The old number of channels

The BBC Microcomputer supported 4 analogue channels, as did the Acorn Electron with the Plus 1 interface.

The BBC Master Compact would allow numbers up to 128 on this call, in reality it supported 2.

30.10.21 OSByte 0x11 (17) Force ADC conversion

Description Force the ADC conversion for a specific channel

On entry

A = `0x11`

X = n Analogue channel to start conversion

$X > 4$ Taken as if $n = 4$

On exit

$X \leq 4$ Preserved

$X > 4$ Set to 4

30.10.22 OSByte 0x12 (18) Reset soft function keys

Description Reset function key definitions

On entry

A = `0x12`

On exit

X Undefined

Y Preserved

Function key definitions are preserved until either a hard break or by this call being made.

30.10.23 OSByte 0x13 (19) Wait for vertical sync

Description Wait until the next frame of the display.

On entry

A = `0x13`

This call will wait until the start of the next frame of the display, which for UK machines occurs 50 times a second.

Its intention is to provide an OS friendly method of synchronizing animations which update the display without any flicker.



Caution

User code trapping `IRQ1` may stop this call from working

30.10.24 OSByte 0x14 (20) Explode character definition RAM

Platform BBC Model B, BBC Model B+ and Acorn Electron

Description Explode character definition RAM

On entry

A = `0x14`

X See Table 30.10

Table 30.10: P.621

On the BBC Model B this call explodes the font definitions from ROM into RAM so that more characters can have their definitions changed.

On systems with a second processor the characters are always fully exploded.

Before OSHWM is changed during a font explosion, a Paged ROM service call is made warning them of the change.

Table 30.10: Character definition expansion

X	ASCII	Start Address	
0	0x80 - 0x9F	0x0C00	imploded
1	0xA0 - 0xBF	OSHWM+0x0100	+above
2	0xC0 - 0xDF	OSHWM+0x0200	+above
3	0xE0 - 0xFF	OSHWM+0x0300	+above
4	0x20 - 0x3F	OSHWM+0x0400	+above
5	0x40 - 0x5F	OSHWM+0x0500	+above
6	0x60 - 0x7F	OSHWM+0x0600	+above

30.10.25 OSByte 0x14 (20) Reset character definitions

Platform BBC Master 128 and BBC Master Compact

Description Reset character definitions

On entry

A = 0x14

On the BBC Master 128 the character definitions are always fully exploded, so this call takes no parameters and resets the character definitions to the defaults.

See also **OSBYTE** 0x19[20] which can reset some but not all characters.

20: **OSBYTE** 0x19 P.624

30.10.26 OSByte 0x15 (21) Flush specific buffer

Description Flush a specific buffer

On entry

A = `0x15`

X See Table 30.11

Table 30.11: P.622

21: **OSBYTE `0x0F`** P.618

Use **OSBYTE `0x0F`**[21] to flush all buffers or just the current input buffer.

Table 30.11: Buffer types for OSBYTE 0x15

X	Buffer to empty
`0`	Keyboard
`1`	Serial Input
`2`	Serial Output
`3`	Printer
`4`	Sound channel 0
`5`	Sound channel 1
`6`	Sound channel 2
`7`	Sound channel 3
`8`	Speech

Errata

The BBC Model B User Guide[10] and the New Advanced User Guide[4] state that the buffer number in *X* is the same as for **OSBYTE `0x80`**[22]. This is incorrect as that call uses negative numbers as it's used with the BASIC **ADVAL** function.

The correct table, reproduced in Table 30.11 is actually on P418 of the original user guide, under **OSBYTE `0x8A`**.

The original Advanced User Guide for the BBC Microcomputer[5], the Acorn Electron Advanced User Guide[8] as well as the BEEB Wiki[11] also show the correct table.

[10] *BBC Micro. User Guide*, (Oct. 1984) P.412

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.143-144

22: **OSBYTE `0x80`** P.638

[5] *Adv. User Guide, BBC Micro*, Bray (Cambridge Microcomputer Center Nov. 1983) P.138

[8] *Adv. User Guide, Electron*, Dickens (Adder Publishing, Cambridge June 2008) P.35

[11] *OSBYTE 0x15*, beebwiki.mdfs.net (Apr. 28, 2026)

30.10.27 OSByte 0x16 (22) Increment polling semaphore

Platform Acorn Electron, BBC Master 128 and BBC Master Compact

Description Increment the polling semaphore

On entry

A = `0x16`

On exit

A Preserved

X Preserved

Y Undefined

This call is only implemented on the BBC Master 128 and Acorn Electron.

When non-zero the operating system will issue a Paged ROM service call with $A = 0x15$ at centisecond intervals.

This utilizes **OSBYTE `0xB9`**[23] on the Acorn Electron, and **OSBYTE `0xB3`**[24] on the BBC Master 128 and BBC Master Compact due to differences in the layout of the system variables.

23: OSBYTE `0xB9` Subsection 30.10.118
24: OSBYTE `0xB3` Subsection 30.10.110

30.10.28 OSByte 0x17 (23) Decrement polling semaphore

Platform Acorn Electron, BBC Master 128 and BBC Master Compact

Description Decrement the polling semaphore

On entry

A = `0x17`

This call complements **OSBYTE `0x16`**[25].

25: OSBYTE `0x16` P.623

30.10.29 OSByte 0x18 (24) Select external Sound

Platform Acorn Electron

Description Select an alternate sound system

On entry

A = `0x18`

On exit

A Preserved

This call is only implemented on the Acorn Electron and is intended to allow for an alternate sound system to be selected.

The value of the X register is dependent on the sound system implementation.

30.10.30 OSByte 0x19 (25) Restore default font definitions

Platform BBC Master 128 and BBC Master Compact

Description Restore group of font definitions back to their defaults

On entry

A = `0x19`

X See Table 30.12

Table 30.12: P.625

This call was introduced on the BBC Master 128 and complements **OSBYTE** `0x14`[26] so that only certain character groups are reset.

26: **OSBYTE** `0x14` P.621

Calling with $X = 0$ is the same as **OSBYTE** `0x14` on the BBC Master 128 as that value resets all character definitions. All other values of X reset a block of 32 characters.

X	ASCII		Hex	
0	32	255	0x20	0xFF
1	32	63	0x20	0x3F
2	64	95	0x40	0x5F
3	96	127	0x60	0x7F
4	128	159	0x80	0x9F
5	160	191	0xA0	0xBF
6	192	223	0xC0	0xDF
7	224	255	0xE0	0xFF

Table 30.12: Reset character definition ranges

30.10.31 OSByte 0x44 (68) Test for sideways RAM

Platform BBC Master 128 and BBC Master Compact

Description Tests for the presence of sideways ram on the BBC Model B+ and BBC Master 128.

On entry

A = 0x44

On exit

X See Table 30.13

Table 30.13: P.626

The test checks to see if a byte within page 0x80 can be written to the four Paged ROM banks that can have either ram or rom. If the memory is writable then it is ram.

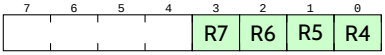


Table 30.13: Sideways RAM allocation

Bit	Allocation
0	Set then ROM 4 is RAM
1	Set then ROM 5 is RAM
2	Set then ROM 6 is RAM
3	Set then ROM 7 is RAM

30.10.32 OSByte 0x45 (69) Sideways RAM allocation

Platform BBC Master 128 and BBC Master Compact

Description Returns how sideways ram on the BBC Model B+ and BBC Master 128 is being used.

On entry

A = `0x45`

On exit

X See Table 30.14

Table 30.14: P.626

Sideways ram on the BBC Master 128 can be used as either Paged ROM or as memory accessible via pseudo addresses.

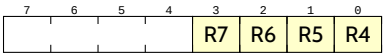


Table 30.14: Sideways RAM Addressing

Bit	Allocation
0	Set then ROM 4 is in extended addressing
1	Set then ROM 5 is in extended addressing
2	Set then ROM 6 is in extended addressing
3	Set then ROM 7 is in extended addressing

30.10.33 OSByte 0x6B (107) Select 1MHz bus/cartridge

Platform BBC Master 128

Description Select $1MHz$ bus – BBC Master 128 only

On entry

A = `0x6B`

X = 0 Select external bus running at $1MHz$

X = 1 Select internal bus running at $2MHz$

The BBC Master 128 has two $1MHz$ buses and this call selects which one is in use.

The default is the one which, like the original BBC Model B is located under the keyboard and,as its name implies, runs at $1MHz$.

The other bus is on the two cartridge slots. Unlike the Acorn Electron, whoms cartridge slots the master's was based on, they run at $2MHz$, and you need to enable them with this call.

The only has the cartridge slot so that is always selected, hence this call is not marked as supported for that platform.

30.10.34 OSByte 0x6C (108) Select screen memory for direct access

Platform BBC Master 128 and BBC Master Compact

Description Select screen memory for direct access

On entry

A = `0x6C`

X = 0 Main memory selected

X = 1 Shadow memory selected

This is only on the BBC Master 128. This call selects which bank of memory is available in the range `0x3000`...`7FFF` to normal programs.

30.10.35 OSByte 0x6D (109) Make temporary file system permanent

Platform BBC Master 128 and BBC Master Compact

Description Make temporary file system permanent

On entry

A = `0x6D`

If this call is issued whilst a temporary filesystem is active, it makes that filesystem permanent.

30.10.36 OSByte 0x70 (112) Select memory for VDU

Platform BBC Master 128 and BBC Master Compact

Description Select main or shadow memory for VDU access

On entry

A = `0x70`

X See Table 30.15

Table 30.15: P.629

This is only available on the BBC Master 128 and operates alongside **OSBYTE `0x71`**[27].

27: OSBYTE `0x71` P.630

This call selects which memory is used by the VDU routines. In normal use VDU uses the same block of memory that is currently being displayed ($X = 0$). However, for more advanced use, *e.g.* rapid animation, you can use this call to select the other block of memory to that actually being displayed to the user. This can then ensure that the display is updated without any flicker caused by updates being issued whilst its being displayed.

X	Memory region	Modes
0	Main memory	0 ... 7
	Shadow memory	128 ... 135
1	Main memory	All modes
2	Shadow memory	All modes

Table 30.15: Memory selection for OSBYTE 0x70 and 0x71

30.10.37 OSByte 0x71 (113) Select memory for display

Platform BBC Master 128 and BBC Master Compact

Description Select main or shadow memory for VDU access

On entry

A = `0x71`

X See Table 30.15

Table 30.15: P.629

This is only available on the BBC Master 128 and operates alongside **OSBYTE `0x70`**[28]. This call selects which memory is displayed to the user.

28: **OSBYTE `0x70`** P.629

30.10.38 OSByte 0x72 (114) Write shadow memory use

Platform BBC Master 128 and BBC Master Compact

Description Force use of shadow memory

On entry

A = `0x72`

X = 0 Shadow memory is always used

$X \neq 0$ Not used for modes 0 ... 7

On the BBC Master 128 and BBC Model B+ shadow memory is only used for modes with bit 7 set – *e.g.* 128 ... 135. This call enables shadow memory to be forced for all modes.

30.10.39 OSByte 0x73 (115) Blank or restore palette

Platform Acorn Electron

Description Blank or Restore colour palette

On entry

A = `0x73`

X = 0 Restore the palette

$X \neq 0$ Set palette to black

On the Acorn Electron will set all colours to either black or the default colours.

30.10.40 OSByte 0x74 (116) Reset sound system

Platform Acorn Electron

Description Reset the sound system on the Electron

On entry

A = `0x74`

This call resets the sound system on the Acorn Electron.

It has no effect on the BBC Model B+, BBC Master 128 or BBC Master Compact.

If used on the BBC Model B a bad command error is generated.

30.10.41 OSByte 0x75 (117) Read VDU status

Description Read current VDU status

On entry

A = `0x75`

On exit

X See Table 30.16

Shadow mode (bit 4) is only used on the BBC Model B+, BBC Master 128 or BBC Master Compact.

Table 30.16: P.632
This is stored at address `0x0x00D0`

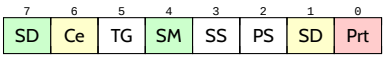


Table 30.16: VDU Status

Bit	Lab	Status
0	Prt	Printer output enabled by VDU 2
1	SD	Scrolling disabled – cursor editing
2	PS	Paged scrolling selected
3	SS	Software scrolling selected – text window
4	SM	Shadow mode selected
5	TG	Text at graphics cursor enabled by VDU 5
6	Ce	Cursor editing mode – cursors separated
7	SD	Screen disabled after VDU 21

30.10.42 OSByte 0x76 (118) Make LEDs reflect keyboard status

Description Reflect keyboard status in LEDs

On entry

A = `0x76`

On exit

X Bit 7 set if Control is pressed

This call ensures that they keyboard LEDs are in the same state as the internal state of the Caps Lock and Shift Lock keys. On the Acorn Electron this only checks the Caps lock as that machine does not have a shift lock key.

30.10.43 OSByte 0x77 (119) Close *SPOOL/*EXEC files

Description Close any `*SPOOL` or `*EXEC` files.

On entry

A = `0x77`

Close any `*SPOOL` or `*EXEC` files.

In addition, a Paged ROM service call with $A = 0x10$ is also generated.

30.10.44 OSByte 0x78 (120) Write current keys pressed

Description Write current keys pressed

On entry

A = `0x78`

This call should only be made by filesystems which have recognized a key being pressed during a restart.

Specifically, when Break is pressed and a file system detects that it's key is pressed, then it should use this call to write the old key pressed value to prevent it being inserted into the keyboard buffer.

30.10.45 OSByte 0x79 (121) Perform keyboard scan

Scan for specific key

Description Scan the keyboard for a specific key

On entry

A = `0x79`

$X \geq 0x80$ Scan specific key

On exit

$X < 0$ Key is pressed

For this, X is the required internal key code listed in Table D.11 ORed with `0x80`.

Table D.11: P.988

Scan from a key

Description Scan the keyboard from a key

On entry

A = `0x79`

X < $0x80$ Scan from key

On exit

X = $0xFF$ No Key is pressed

X ≠ $0xFF$ Internal Key number pressed

Except for the BBC Model B the keys are scanned in ascending order.

On the BBC Model B they are scanned in the order:
$0x10, 0x20 \ldots 0x70, 0x21 \ldots 0x71$, *etc.*$0x29 \ldots 0x79$

The key code used as the start is the internal key code listed in Table D.11.

Table D.11: P.988

30.10.46 OSByte 0x7A (122) Perform keyboard scan from 0x10

Description Scan the keyboard from key 0x10

On entry

A = `0x7A`

On exit

X = $0xFF$ No Key is pressed

X ≠ $0xFF$ Internal Key number pressed

This is equivalent to OSBYTE `0x79` with $X = 0x10$

30.10.47 OSByte 0x7B (123) Printer driver going dormant

Description Notify that printer driver is going dormant

On entry

A = `0x7B`

On exit

A, X Preserved

Y Preserved on I/O processor

Y Not passed across the tube

When a printer driver goes dormant, it should make this call. The operating system will need to wake up the driver when more characters are placed into the printer buffer.

30.10.48 OSByte 0x7C (124) Clear ESCAPE condition

Description Clear ESCAPE condition with no further action

On entry

A = `0x7C`

On exit

X = 0 No escape condition to clear

X = $0xFF$ Escape condition cleared

Clears any escape condition without any further action. The Tube is notified if active.

See **OSBYTE `0x7E`**[29] on clearing the condition with side effects

29: **OSBYTE `0x7E`** P.637

30.10.49 OSByte 0x7D (125) Set ESCAPE condition

Description Set ESCAPE condition

On entry

A = `0x7D`

On exit

A, X, Y Preserved

Partially emulate the pressing of the Escape key. The Tube is notified if active. An Escape event is not generated.

30.10.50 OSByte 0x7E (126) Acknowledge ESCAPE, clear + effects

Description Clear ESCAPE condition with effects

On entry

A = `0x7E`

On exit

$X = 0$ No escape condition to clear

$X = 0xFF$ Escape condition cleared

This call attempts to clear the escape condition by flusing all active buffers, close any `*EXEC` files, reset the VDU paging counter and queue, stop any soft key expansion and stop any sounds.

See **OSBYTE `0x7C`**[30] for clearing Escape without any side effects.

30: OSBYTE `0x7C` P.636

30.10.51 OSByte 0x7F (127) Check for EOF

Description Check for file EOF

On entry

A = `0x7F`

X File handle of the file to check

On exit

X ≠ 0 End of the file has been reached

X = 0 End of the file has not been reached

30.10.52 OSByte 0x80 (128) Read ADC channel/buffer status

X = 0

Description Read analog port

On entry

A = `0x80`

X = 0

On exit

X Fire button status

Y Last ADC channel conversion

The X register holds the fire button status in bits 0 and 1.

The Y register holds the last channel which has had an ADC conversion performed. This is the channel number 1 ... 4.

This is the equivalent of the BBC Basic **ADVAL** (0) function.

X > 0

Description Read analog value from ADC Channel

On entry

A = `0x80`

$X \geq 1$ ADC Channel to sample

On exit

XY 16-bit value for the channel;

This is the BBC Basic **ADVAL** function. The ADC channel $1 \ldots 4$ is passed in the X register and the result is returned in both the X and Y registers. X holds the low 8-bits whilst Y the upper 8-bits of the result.

On the BBC Master Compact, only two channels are provided, so channels 1,3,5 and 7 are the same, whilst 2,4,6 and 8 are the other channel. If the sprite pointer ROM is fitted then channels 7 and 8 are used for the pointer movements.

X < 0

Description Read buffer status

On entry

A = `0x80`

$X < 0$ See Table 30.17

Table 30.17: P.640

On exit

X Buffer status

For input buffers X returns the number of characters in the buffer.

For output buffers X returns the number of spaces remaining in the buffer before it is full.

The result for a buffer not implemented on the Acorn Electron will be meaningless.

Table 30.17: Query Buffer Status

X		Buffer
FF	255	Keyboard
FE	254	RS-423 Input
FD	253	RS-423 Output
FC	252	Printer
FB	251	Sound channel 0
FA	250	Sound channel 1
F9	249	Sound channel 2
F8	248	Sound channel 3
F7	247	Speech

[10] *BBC Micro. User Guide*, (Oct. 1984) P.412

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.143-144

31: **OSBYTE `0x15`** P.622

Errata

The BBC Model B User Guide[10] and the New Advanced User Guide[4] wrongly link **OSBYTE `0x15`**[31] to this table for its entry parameters.

30.10.53 OSByte 0x81 (129) Read key with time limit

Description Read a key within a time limit

On entry

A = `0x81`

XY Time limit

On exit Key pressed

X ASCII character of key

Y = 0

$Carry$ = 0

On exit Error condition

X Undefined

Y = `0xFF` Timeout

Y = `0x1B` Escape pressed

$Carry$ = 1

The maximum timeout is `0x7FFF` centiseconds, which is about $5m27s$.

This is due to Y = `0xFF` being used for the alternate actions for this **OSBYTE** call.

30.10.54 OSByte 0x81 (129) Scan for specific key

Description Scan keyboard for key press

On entry

A = `0x81`

$X \neq$ `0x00` Negative INKEY value of key

Y = `0xFF`

On exit Key pressed

X = `0xFF`

Y = `0xFF`

$Carry$ = 1

On exit Key not pressed

X = `0x00`

Y = `0x00`

$Carry$ = 0

The INKEY values are listed in Table D.11

Table D.11: P.988

30.10.55 OSByte 0x81 (129) Read machine type

Description Read the OS version number.

On entry

A = `0x81`

X = 0

Y = `0xFF`

On exit

X See Table D.10

Table D.10: P.983

This is the equivalent to **INKEY(-256)** in BBC Basic.

OSBYTE `0x00` returns a slightly different list of versions in Table 30.3

Table 30.3: P.608

BBC Basic runs on far more platforms than those made by Acorn, so on modern machines this call can return far more than the original entries. Those entries are included in Table D.10.

Errata

The New Advanced User Guide[4] shows different values for the OS version for the BBC Master Compact. In Table 30.3 for **OSBYTE `0x00`**, the version is 5.0. However, in Table D.10 the version is 5.10.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.428

Table 30.3: P.608

For the German MOS, all documentation states this returns `0xFC`. However, a footnote in the BeebWiki[12] entry indicates that all known German versions of MOS return `0xFF`.

[12] *INKEY*, beebwiki.mdfs.net (May 10, 2026)

30.10.56 OSByte 0x82 (130) Read machine high order address

Description Return the machine high order address

On entry

A = `0x82`

On exit

XY Top 16-bits of the address

This returns the upper 16-bits of the address used by filesystems which use 32-bit addresses. X holds the lower 8-bit whilst Y the upper 8-bits.

30.10.57 OSByte 0x83 (131) Read top of OS RAM

Description Returns OSHWM

On entry

A = `0x83`

On exit

XY OS High Water Mark

`OSHWM` is the address of the first byte of main memory that has not been allocated by the operating system for either its own use or by Paged ROMs.

On exit, X is the low byte whilst Y is the high byte. As `OSHWM` always lies on a page boundary, *e.g.* `0x0E00`, X is always 0.

30.10.58 OSByte 0x84 (132) Read top of user RAM

Description Returns HIMEM on IO processor

On entry

A = `0x84`

On exit

XY HIMEM

`HIMEM` is the address of the first byte of main memory after user RAM. This is usually the first byte of screen memory or `0x8000` when a shadow screen mode is in use.

On exit, X is the low byte whilst Y is the high byte.

On a second processor, `HIMEM` is unrelated to this address

30.10.59 OSByte 0x85 (133) Read HIMEM for given mode

Description Returns HIMEM for a specific mode

On entry

A = `0x85`

X Screen mode number

On exit

XY HIMEM address for that screen mode

`HIMEM` is the address of the first byte of main memory after user RAM. This is usually the first byte of screen memory or `0x8000` when a shadow screen mode is in use.

On exit, X is the low byte whilst Y is the high byte.

30.10.60 OSByte 0x86 (134) Read input cursor pos

Description Return the current text cursor position

On entry

A = `0x86`

On exit

X X character position, POS

Y Y character position, VPOS

30.10.61 OSByte 0x87 (135) Read char at text cursor and screen mode

Description Return the character at the text cursor and the current screen mode

On entry

A = `0x87`

On exit

X ASCII character code

Y Mode number

The mode number returned in Y is always 0 ... 7 regardless if shadow modes are actually in use.

30.10.62 OSByte 0x88 (136) *CODE

Description Execute USERV code

On entry

A = 0x88

On exit

X, Y Passed to **USERV** routine

This implements the ***CODE**[32] command. When invoked it will call the routine addressed by the **USERV** vector with $A = 0$ and the X and Y registers from this call.

32: ***CODE** P.777

30.10.63 OSByte 0x89 (137) *MOTOR

Description Turn the cassette motor relay on or off

On entry

A = 0x89

On exit

X = 0 Relay off

X = 1 Relay on

This implements the ***MOTOR**[33] command.

33: ***MOTOR** P.778

30.10.64 OSByte 0x8A (138) Insert into buffer

Description Insert character into a specific buffer

On entry

A = `0x8A`

X Buffer See Table 30.11

Y Character to insert

On exit

Carry = 0 Value inserted

Carry = 1 Not inserted

The usual cause of an entry not being inserted is if the buffer is full.

Table 30.11: P.622

Errata

The New Advanced User Guide[4] does not show the correct buffer IDs, with the incorrect ones for **OSBYTE `0x80`**[34] above it on the same page.

The BEEB Wiki[13] does not show the table either, nor mentions `0x0x15`, but it does list $X = 9$ which is for RISC-OS.

The correct table, reproduced in Table 30.11 on page 622 is present on the page for this actual call in the original BBC Microcomputer User Guide[10].

The original Advanced User Guide for the BBC Microcomputer[5] and the Acorn Electron Advanced User Guide[8] show the correct table.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.144

34: OSBYTE `0x80` P.638

[13] *OSBYTE 0x8A*, beebwiki.mdfs.net (Apr. 28, 2026)

[10] *BBC Micro. User Guide*, (Oct. 1984) P.418

[5] *Adv. User Guide, BBC Micro*, Bray (Cambridge Microcomputer Center Nov. 1983) P.162

[8] *Adv. User Guide, Electron*, Dickens (Adder Publishing, Cambridge June 2008) P.51

30.10.65 OSByte 0x8B (139) *OPT

Description Select filesystem options

On entry

A = `0x8B`

X First parameter

Y Second parameter

On exit

A Preserved

X, *Y* Undefined

This implements the *OPT[35] command. The values of *X* and *Y* are specific to the current active filesystem.

Table 30.18 on page 649 shows the values for the TAPE and ROM filing systems. May other filesystems use this as a basis for their options.

X = 3 in this example is only for the TAPE filesystem.

35: *OPT P.??

X	Y	Setting
`0`	`0`	restore *OPT
`1`	`0`	turn off fs messages
`1`	`1`	turn on fs messages
`1`	`2`	turn on extended messages
`2`	`0`	ignore errors, may message
`2`	`1`	on error prompt for re-try
`2`	`2`	on error abort
`3`	`n`	Interblock gaps in n/10 seconds

Table 30.18: *OPT for TAPE and ROM filesystems

30.10.66 OSByte 0x8C (140) *TAPE

Description Activate the TAPE filesystem

On entry

A = `0x8C`

X See Table 30.19

Table 30.19: P.650

On exit

X = 0 Relay off

X = 1 Relay on

36: ***TAPE** P.778
37: TAPE P.787

This implements the ***TAPE**[36] command which activates the TAPE[37] filesystem.

On the Acorn Electron this only switches filesystem as it only supports 1200 baud.

Table 30.19: Cassette Speeds

X	Baud rate
`0`	1200 baud Default
`3`	300 baud
`12`	1200 baud

30.10.67 OSByte 0x8D (141) *ROM

Description Activate the ROM filesystem

On entry

A = `0x8D`

38: ***ROM** P.??
39: ROM P.789

This implements the ***ROM**[38] command which activates the ROM[39] filesystem.

30.10.68 OSByte 0x8E (142) Enter language ROM

Description Enter language rom

On entry

A = `0x8E`

X Paged ROM number

This call will select the requested Paged ROM number, print out the language name and then enter it via the ROMs language entry point with $A = 1$.

Locations `0x00FD` and `0x00FE` will be set with the copyright message in the ROM.

Second Processor

When a Second Processor is used, the rom will first be copied over the Tube, potentially to a relocation address if the ROM supports it before it is entered.

30.10.69 OSByte 0x8F (143) Issue paged ROM service request

Description Issue Paged ROM service request

On entry

A = `0x8F`

X Reason code

Y Parameter

On exit

X = 0 A ROM claimed the call

Y May contain return argument

30.10.70 OSByte 0x90 (144) *TV

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Description Configure TV output

On entry

A = `0x90`

$X > 0$ Shift screen up

$X < 0$ Shift screen down

$Y = 0$ Interlace on

$Y = 1$ Interlace off

On exit

X Old screen shift

Y Old interlace option

40: *TV P.779

This implements the *TV[40] command which configures the TV output.

The value of X is the number of screen lines to move, in two's complement. Hence, 1 would move the picture up one line whilst 255 down one line.

30.10.71 OSByte 0x90 (144) *TV

Platform Acorn Electron

Description Does nothing

On entry

A = `0x90`

The RF output on the Acorn Electron cannot be congfigured so this call does nothing.

30.10.72 OSByte 0x91 (145) Get character from buffer

Description Get a character from a buffer

On entry

A = `0x91`

X Buffer See Table 30.11

Table 30.11: P.622

On exit

$Carry$ = 0 Char removed

$Carry$ = 1 Buffer empty

Y Character removed

Y is only valid when $C = 0$. When $C = 1$ the buffer was empty.

Errata

The New Advanced User Guide[4] does not show any buffer IDs for this call.

The BEEB Wiki[14] does not show the table either.

The original BBC Microcomputer User Guide[10] lists just two, $X = 0$ for the keyboard and $X = 1$ for RS-423.

The original Advanced User Guide for the BBC Microcomputer[5] and the Acorn Electron Advanced User Guide[8] mention the full buffer list in **OSBYTE `0x15`**[41] listed here in Table 30.11 on page 622.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.144

[14] *OSBYTE 0x91*, beebwiki.mdfs.net (Apr. 28, 2026)

[10] *BBC Micro. User Guide*, (Oct. 1984) P.420

[5] *Adv. User Guide, BBC Micro*, Bray (Cambridge Microcomputer Center Nov. 1983) P.169

[8] *Adv. User Guide, Electron*, Dickens (Adder Publishing, Cambridge June 2008) P.53

41: **OSBYTE `0x15`** P.622

30.10.73 OSByte 0x92 (146) Read from FRED

Description Read byte from FRED

On entry

A = `0x92`

X Offset in page

On exit

Y Byte read

FRED is page `0xFC` in the BBC Microcomputer and contains usually devices accessible over the 1*MHz* bus.

Although a lot of software wrote directly to this page, that was only possible when the code ran on the main, I/O processor. When a second processor was used this was not possible.

That was where this call, and it's sibling made possible as the call would be sent over the Tube.

Table 30.20: Current FRED allocations

From	To	1*MHz* Bus	Cartridge
`FC00`	`FC0F`	Test Hardware	
`FC10`	`FC13`	Teletext	
`FC14`	`FC1F`	Prestel	Reserved
`FC20`	`FC27`	IEEE 488	
`FC28`	`FC2F`	Econet on Electron	
`FC30`	`FC3F`	Cambridge Ring	
`FC40`	`FC47`	Winchester Disc	
`FC48`	`FC4F`	Reserved	
`FC60`	`FC6F`	Serial Expansion	
`FC80`	`FC8F`	Test Hardware	Test hardware
`FCB0`	`FCBF`	6522 VIA on Electron	
`FCC0`	`FCCF`	1770 FDC on Electron	
`FCE0`	`FCEF`	Tube on Electron	
`FCFF`		JIM paging register	JIM paging

30.10.74 OSByte 0x93 (147) Write to FRED

Description Write a byte to FRED

On entry

A = `0x93`

X Offset in page

Y Byte to write

30.10.75 OSByte 0x94 (148) Read from JIM

Description Read byte from JIM

On entry

A = `0x94`

X Offset in page

On exit

Y Byte read

JIM is page `0xFD` in the BBC Microcomputer and, along with the paging register in FRED, provides access to up to 64K of additional memory.

30.10.76 OSByte 0x95 (149) Write to JIM

Description Write a byte to JIM

On entry

A = `0x95`

X Offset in page

Y Byte to write

30.10.77 OSByte 0x96 (150) Read from SHIELA

Description Read byte from SHIELA

On entry

A = `0x96`

X Offset in page

On exit

Y Byte read

SHIELA is page `0xFE` in the BBC Microcomputer and contained hardware connected directly to the computer.

Although a lot of software wrote directly to this page, that was only possible when the code ran on the main, I/O processor. When a second processor was used this was not possible.

That was where this call, and it's sibling made possible as the call would be sent over the Tube.

30.10.78 OSByte 0x97 (151) Write to SHIELA

Description Write a byte to SHIELA

On entry

A = `0x97`

X Offset in page

Y Byte to write

30.10.79 OSByte 0x98 (152) Examine buffer status

Platform Acorn Electron, BBC Master 128 and BBC Master Compact

Description Examine status and return next value in buffer but does not remove it



For earlier versions of MOS, *e.g.* BBC Model B see the following page

Table 30.11: P.622

On entry

A = `0x98`

X See Table 30.11

On exit Buffer not empty

Y Next character in buffer

Carry = 0

A, *X* Preserved

On exit Buffer empty

Y Preserved

Carry = 1

A, *X* Preserved

Unlike on the BBC Model B, this call does not need to be called with interrupts disabled. It also returns the next character without removing it in the Y register

30.10.80 OSByte 0x98 (152) Examine buffer status



This is for MOS up to and including 1.20. For later machines see the previous page

Table 30.11: P.622

Platform BBC Model B

Description Get pointer to next char to read in buffer

On entry

A = `0x98`

X See Table 30.11

On exit

Y Index to next character

Carry = 0

ZPage `0x00FA` address of buffer

A, *X* Preserved

On exit

Y Preserved

Carry = 1

A, *X* Preserved

After using this call, if you need to read but not remove the next character, to use the address in `0x00FA` with the index in the Y register.

You should disable interrupts whilst making this call and reading from the buffer to ensure that no interrupt updates the buffer.

Listing 30.2 on page 659 shows a simple example.

OSBYTE 0x98 example

In this example we disable maskable interrupts before calling **OSBYTE 0x98** to query the keyboard buffer.

If the buffer was not empty we then get the current character into the accumulator.

We then re-enable maskable interrupts before returning to the caller.

The calling code can then test the carry flag and if it was clear then the accumulator would hold the next character in the buffer that would be removed from the buffer.

Listing 30.2: Using OSBYTE 0x98 on the BBC B

```
                        ; Example of peeking the keyboard buffer
                        ; on the BBC Model B.

                        ; This call is completely different on
                        ; on the Electron or Master
.PEEKBUF
         SEI            ; Disable interrupts
         LDA #0x98      ; OSBYTE 0x98
         LDX #0         ; Keyboard buffer
         JSR OSBYTE
         BCS EXIT       ; Buffer empty?
         LDA (0xFA),Y   ; Read char
                        ; Enable interrupts before
                        ; returning to the caller
.EXIT
         CLI
         RTS
```

30.10.81 OSByte 0x99 (153) Insert character into input buffer

Description Insert a character into the input buffer. Handle ESCAPE event as necessary.

On entry

A = `0x99`

X = 0 Keyboard buffer

X = 1 RS-423 buffer

Y Character to insert

On exit

A Preserved

X, Y Undefined

If the selected input buffer is full then the Input buffer full event (type 1) is issued.

When a character is inserted into an input buffer it is tested to see if either an input event or an escape event should be issued.

An Input event (type 2) is generated if all of the following apply:

- The character is not the ESCAPE[42] character
- It is the ESCAPE character but are to be treated as normal[43] characters
- Escapes are protected[44]
- If RS-423 input is enabled and $X = 1$ then RS-423 ESCAPEs are suppressed. This makes this call identical to **OSBYTE `0x8A`**[45]. See **OSBYTE `0xB5`**[46] for the RS-423 escape mode.

If the above did not apply then the ESCAPE event (type 6) is issued.

Table 30.9 on page 617 lists the event types.

42: ESCAPE P.715

43: normal P.724

44: protected P.699

45: **OSBYTE `0x8A`** P.648

46: **OSBYTE `0xB5`** P.682

30.10.82 OSByte 0x9A (154) Write to video ULA

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Description On the BBC Model B and BBC Master 128 this writes to register 0 of the video ULA and updates the ram copy of that register.

On entry

A = `0x9A`

X Value to write

On exit

A, X, Y Preserved

30.10.83 OSByte 0x9A (154) Reset flash cycle

Platform Acorn Electron

Description On the Acorn Electron this resets the flash cycle to the beginning of the mark state.

On entry

A = `0x9A`

30.10.84 OSByte 0x9B (155) Write to video ULA palette register

BBC B and Master

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Description On the BBC Model B and BBC Master 128 this writes to register 1 of the video ULA and updates the ram copy of that register.

On entry

A = `0x9B`

X Value to write

On exit

A, X, Y Preserved

The actual value written to both the register and memory is:

$$X \; EOR \; 7$$

30.10.85 OSByte 0x9B (155) No operation

Platform Acorn Electron

Description This is unimplemented on the Acorn Electron and simply executes RTS.

On entry

A = `0x9B`

30.10.86 OSByte 0x9C (156) Read/Write 6850 control register

Platform Acorn Electron

Description On the BBC Model B and BBC Master 128 this can be used to change the 6850 ACIA control register.

On entry

A = `0x9C`

A = `0x`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location



Table 30.21 on page 664 shows bits which affect the baud rates. They also affect the cassette[47] baud rate on the BBC Microcomputer.

The default for RS-423 is `10` for divide by 64.

47: Not on the Acorn Electron as that's handled by the ULA

Table 30.22 on the next page shows the bits which affect the data format. Most modern communications use 8N1 `101`.

Table 30.23 on the following page details the RTS line, Break signal, and the interrupt generated when the transmit data register is empty.

Bit 7 enables interrupts on receive data register full, over run or DCD transition.

Table 30.21: 6850 ACIA control register bits 0...1

1	0	Effect
0	0	Divide counter by 1
0	1	Divide counter by 16
1	0	Test Divide counter by 64
1	1	Master reset

Table 30.22: 6850 ACIA control register bits 2...4

4	3	2	Word size	Parity	Stop Bits	Abbrev
0	0	0	7	E	2	7E2
0	0	1	7	O	2	7O2
0	1	0	7	E	1	7E1
0	1	1	7	O	1	7O1
1	0	0	8	E	2	8N2
1	0	1	8	O	2	8N1
1	1	0	8	E	1	8E1
1	1	1	8	O	1	8O1

Table 30.23: 6850 ACIA control register bits 5...6

6	5	RTS	TX Interrupt	Break Level
0	0	low	disabled	
0	0	low	enabled	
0	1	high	disabled	
0	1	low	disabled	on Data output

30.10.87 OSByte 0x9D (157) Fast Tube BPUT

Description A faster alternative to **OSBPUT**[48] when used from a Second Processor.

48: **OSBPUT** P.599

On entry

A = `0x9D`

X Byte to write

Y File handle

On exit

X, *Y* Undefined

30.10.88 OSByte 0x9E (158) Read from speech processor

Platform BBC Model B, BBC Model B+ and Acorn Electron

Description Read from the Speech Processor

On entry

A = `0x9E`

On exit

Y Byte read

This is only supported on the BBC Model B. The Acorn Electron issues a page rom service call.

The byte read is either from the status register or the data register of the Speech Processor, depending on the previous command sent to the Processor.

To read the Speech Processor's data register you should send a "Read Byte" command with **OSBYTE `0x9F`**[49] before making this call. If the "Read Byte" command has not been sent, the status register will be read.

49: **OSBYTE `0x9F`** P.667

30.10.89 OSByte 0x9E (158) Reserved

Platform BBC Master 128 and BBC Master Compact

Description Reserved

On entry

A = `0x9E`

30.10.90 OSByte 0x9F (159) Write to speech processor

Platform BBC Model B and BBC Model B+

Description Write to the Speech Processor

On entry

A = `0x9F`

Y See Table 30.24

Table 30.24: P.667

This is only supported on the BBC Model B.

The Acorn Electron issues a page rom service call.

The speech commands in Table 30.24 on page 667 are listed in the Acorn Speech User Guide[15].

[15] *BBC Micro. Speech User Guide*, (May 4, 1983) P.22

Bits labeled '`a`' represent an address. A '`x`' the processor does not care.

Y	Operation
x000xxxx	NOP
x001xxxx	Read byte
x010xxxx	NOP
x011xxxx	Read and Branch
x100aaaa	Load address
x101xxxx	Speak
x110xxxx	Speak External
x111xxxx	Reset

Table 30.24: Speech processor commands

Errata

For some reason the "Speak External" command x110xxxx is listed out of order. Hopefully this is just that, but with no other reference available care should be made.

The New Advanced User Guide[4] mentions it is not implemented on the BBC Master 128 but does not state what would happen if it was called.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.374

30.10.91 OSByte 0x9F (159) Reserved

Platform BBC Master 128 and BBC Master Compact

Description Reserved

On entry

A = 0x9F

30.10.92 OSByte 0xA0 (160) Read VDU variable

Platform BBC Model B+, Acorn Electron, BBC Master 128 and BBC Master Compact

Description Read VDU variable

On entry

A = `0xA0`

X VDU Variable ID

On exit

X Variable low byte

Y Variable high byte

Errata

This call is not listed in the BBC Model B User Guide[10].

In the New Advanced User Guide[4] it mentions that the variables are defined in the BBC Master Reference Guide Part 1[9].

However, the Acorn Electron Advanced User Guide[8] mentions that it's implemented but is officially undefined and may change in the future.

So care should be taken when using this call anywhere other than on the BBC Master 128.

[10] *BBC Micro. User Guide*, (Oct. 1984) P.412

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.183

[9] *Master Series Ref. Manual Part 1*, (Jan. 1986) E.4-1

[8] *Adv. User Guide, Electron*, Dickens (Adder Publishing, Cambridge June 2008) P.57

30.10.93 OSByte 0xA1 (161) Read CMOS RAM

Platform BBC Master 128

Description Read from CMOS RAM

On entry

A = `0xA1`

X See Table D.13

Table D.13: P.993

On exit

X Corrupt

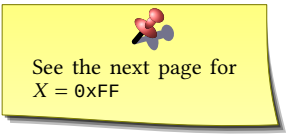
Y byte read

The BBC Master 128 as a small CMOS Static RAM chip to hold its configuration. On entry X can be a value between 0 ... 49.

30.10.94 OSByte 0xA1 (161) Read EEPROM

Platform BBC Master Compact

Description Read from EEPROM



On entry

A = `0xA1`

X ≠ `0xFF` See Table D.13

Table D.13: P.993

On exit

X Corrupt

Y byte read

The BBC Master Compact has a EEPROM chip to store its configuration. On entry X can be a value between either 0 ... 127 or 0 ... 254 dependent on the size of the EEPROM.

30.10.95 OSByte 0xA1 (161) Query EEPROM size

Platform BBC Master Compact

Description Query EEPROM size

On entry

A = `0xA1`

X = `0xFF`

On exit

X Corrupt

Y = 0 No EEPROM

Y = `0x7F` 128 byte EEPROM

Y = `0xFF` 256 byte EEPROM

30.10.96 OSByte 0xA2 (162) Write CMOS RAM/EEPROM

Platform BBC Master 128 and BBC Master Compact

Description Write to CMOS RAM or EEPROM

On entry

A = `0xA2`

X See Table D.13

Y Value to write

Table D.13: P.993

On exit

X Preserved

Y Corrupt

30.10.97 OSByte 0xA3 (163) Enable or Disable printer and ADC

Platform Acorn Electron

Description Enable or Disable both the printer and ADC

On entry

A = `0xA3`

X = 128

Y = 0 Enable

Y = 1 Disable

On a standard machine[8] it issues a Paged ROM service request but the Plus 1 interface uses this to enable or disable the printer and ADC conversions.

[8] *Adv. User Guide, Electron*, Dickens (Adder Publishing, Cambridge June 2008) P.57

30.10.98 OSByte 0xA3 (163) Reserved for application software

Platform BBC Master 128 and BBC Master Compact

Description Reserved for application software

On entry

A = `0xA3`

Errata

In the New Advanced User Guide[4] it's listed only in the OSBYTE index with the title "Reserved for application software". It does not list the Acorn Electron version at all.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.438

30.10.99 OSByte 0xA4 (164) Check for 6502 code

Platform BBC Master 128 and BBC Master Compact

Description Checks a language rom to see if it can run on the 6502

On entry

A = `0xA4`

XY Address of code to check

This will never return if the code cannot be executed as it issues a BRK with the appropriate error.

This call performs the following checks:

- Check code is in a Paged ROM by looking for the copyright string at offset `0x07`.
- Check the processor type at offset `0x06`. If bit 7 is clear then a BRK is generated with the error "This is not a language"
- If the processor type bits 0 ... 3 indicate the ROM is not 6502 code or 6502 BASIC then a brk is executed with the error "I cannot run this code"

30.10.100 OSByte 0xA5 (165) Read output cursor position

Platform BBC Master 128 and BBC Master Compact

Description Returns the output cursor position.

On entry

A = `0xA5`

On exit

X Horizontal character position

Y Vertical character position

This is identical to **OSBYTE `0x86`**[50] except when copying text on-screen when this returns the output cursor rather than the text cursor.

50: **OSBYTE `0x86`** P.646

30.10.101 Osbyte 0xA6 (166) Base of MOS variables

Function Base of MOS variables

Variable `0x0236`

Length 2 bytes

On entry

A = `0xA6`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.102 Osbyte 0xA8 (168) ROM Pointer table

Function ROM Pointer table

Variable `0x0238`

Length 2 bytes

On entry

A = `0xA8`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This table of extended vectors consists of 3 byte vectors in for format: Address-low, Addresss-high and ROM-number.

When used over the tube, the address refers to the I/O Processor's memory.

30.10.103 Osbyte 0xAA (170) ROM Information table

Function ROM Information table

Variable `0x023A`

Length 2 bytes

On entry

A = `0xAA`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

When used over the tube, the address refers to the I/O Processor's memory.

The address returned is a 16-byte table containing one byte per Paged ROM. This byte contains the ROM type from address `0x8006` in the ROM or 0 if a valid ROM is not present.

30.10.104 Osbyte 0xAC (172) Keyboard Translation Table

Function Keyboard Translation Table

Variable `0x023C`

Length 2 bytes

On entry

A = `0xAC`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location



The use of this call is not recommended

The table is hardware specific due to the different keyboard matrices used on each machine.

30.10.105 Osbyte 0xAE (174) VDU Variables Table

Function VDU Variables Table

Variable `0x023E`

Length 2 bytes

On entry

A = `0xAE`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

When used over the tube, the address refers to the I/O Processor's memory.

30.10.106 Osbyte 0xB0 (176) CFS timeout counter

Function CFS timeout counter

Variable `0x0240`

On entry

A = `0xB0`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This counter is decremented once every vertical sync. pulse which occurs 50 times per second. The timeout counter is used to time the inter-block gaps and the leader tones on the cassette interface.

30.10.107 Osbyte 0xB1 (177) Input source

Function Input source

Variable `0x0241`

On entry

A = `0xB1`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This is used by **OSBYTE `0x02`**[51]. It should not be used as a replacement as it does not enable various interrupts the main call does.

51: OSBYTE `0x02` Subsection 30.10.6

30.10.108 Osbyte 0xB2 (178) Keyboard semaphore

Function Keyboard semaphore

Variable `0x0242`

On entry

A = `0xB2`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Setting this to `0x00` disables the keyboard interrupt, whilst `0xFF` enables the interrupt. When disabled, the OS does not scan the keyboard.

The Acorn Electron Advanced User Guide[8] also states that disabling the interrupts can cause the machine to run significantly faster - but with the downside that the keyboard has to be scanned by the application as necessary.

[8] *Adv. User Guide, Electron*, Dickens (Adder Publishing, Cambridge June 2008) P.60

30.10.109 Osbyte 0xB3 (179) Default OSHWM

Function Default OSHWM

Platform BBC Model B, BBC Model B+ and Acorn Electron

Variable `0x0243`

Description The address representing the top of the operating system memory regardless user font definitions.

On entry

A = `0xB3`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.110 Osbyte 0xB3 (179) Paged ROM polling semaphore

Function Paged ROM polling semaphore

Platform BBC Master 128 and BBC Master Compact

Variable `0x0243`

Description When this is non-zero, the operating system will issue a Paged ROM service call with $A = 0x15$ at centisecond intervals.

On entry

A = `0xB3`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Managed by **OSBYTE `0x16`**[52] and **OSBYTE `0x17`**[53].

52: **OSBYTE `0x16`** Subsection 30.10.27
53: **OSBYTE `0x17`** Subsection 30.10.28

Errata

The New Advanced User Guide[4] only shows the variable for the Acorn Electron – specifically it shows the **OSBYTE `0xB9`** call but does not mention this one. The BBC Master 128 and BBC Master Compact cannot use the other address as it's used for the Video ULA.

4 *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.326

30.10.111 Osbyte 0xB4 (180) Current OSHWM

Function Current OSHWM

Variable `0x0244`

On entry

A = `0xB4`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This contains the address of the first byte of main memory that has not been allocated by the operating system, or allocated for use by Paged ROMs.

The value returned is the upper byte of the address as this is always on a page boundary so the lower byte is always 0.

30.10.112 Osbyte 0xB5 (181) RS-423 mode

Function RS-423 mode

Variable `0x0245`

On entry

A = `0xB5`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Flag=0 causes input from RS-423 to be treated as if it was the keyboard. ESCAPEs are recognized, soft keys are expanded, the character entering input buffer event is generated and cursor editing is performed.

Flag=1 **Default**: input from RS-423 is handled independently of the keyboard. All characters from RS-423 enter input buffer but the character entering input buffer event is not generated.

30.10.113 Osbyte 0xB6 (182) Character Explosion State

Function Character Explosion State

Platform BBC Model B, BBC Model B+ and Acorn Electron

Variable `0x0246`

On entry

A = `0xB6`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This returns the state of font explosion as set by **OSBYTE `0x14`**[54]

54: OSBYTE `0x14` Subsection 30.10.25

30.10.114 Osbyte 0xB6 (182) Printer Ignore Character

Function Printer Ignore Character

Platform BBC Master 128 and BBC Master Compact

Variable `0x0246`

On entry

A = `0xB6`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Bit 7 is set if the printer ignore character is not in use

30.10.115 Osbyte 0xB7 (183) TAPE or ROM Switch

Function TAPE or ROM Switch

Variable `0x0247`

On entry

A = `0xB7`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

55: *TAPE Section 31.5
56: *ROM ??

This flag is either 0 for the cassette *TAPE[55] filing system, and 2 for the ROM *ROM[56] file system.

Any other value is meaningless.

30.10.116 Osbyte 0xB8 (184) Video ULA control register

Function Video ULA control register

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Variable `0x0248`

On entry

A = `0xB8`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.117 Osbyte 0xB9 (185) Video ULA palette register

Function Video ULA palette register

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Variable `0x0249`

On entry

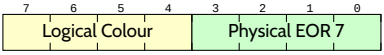
A = `0xB9`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location



Bits	State
0 ... 3	Physical colour EOR 7
4 ... 7	Logical colour

Table 30.25: Video ULA Palette Register

30.10.118 Osbyte 0xB9 (185) Paged ROM polling semaphore

Function Paged ROM polling semaphore

Platform Acorn Electron

Variable `0x0249`

Description When this is non-zero, the operating system will issue a Paged ROM service call with $A = 0x15$ at centisecond intervals.

On entry

A = `0xB9`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Managed by **OSBYTE `0x16`**[57] and **OSBYTE `0x17`**[58].

57: **OSBYTE `0x16`** Subsection 30.10.27
58: **OSBYTE `0x17`** Subsection 30.10.28

Errata

The New Advanced User Guide[4] only shows this variable but does not indicate that it is for just the Acorn Electron.

The BBC Master 128 and BBC Master Compact use a different variable address, specifically **OSBYTE `0xB2`** call, because that call is used for the Video ULA.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.326

30.10.119 Osbyte 0xBA (186) ROM active on last BRK

Function ROM active on last BRK

Variable `0x024A`

On entry

A = `0xBA`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

When a BRK instruction is executed, the operating system records the number of the Paged ROM that was active when it occurred.

This is sometimes useful if the rom was not the same as the current language rom which usually performs error handling.

30.10.120 Osbyte 0xBB (187) BASIC ROM number

Function BASIC ROM number

Variable `0x024B`

On entry

A = `0xBB`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

BBC Basic is recognized as it's a language ROM which does not possess a service entry.

When the ***BASIC** command is executed, it uses this rom number as the entry point.

If no BBC Basic ROM is present then this contains `0xFF`.

30.10.121 Osbyte 0xBC (188) Current ADC channel being sampled

Function Current ADC channel being sampled

Variable `0x024C`

On entry

A = `0xBC`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Returns the ADC channel currently being converted.

30.10.122 Osbyte 0xBD (189) Maximum ADC channel number

Function Maximum ADC channel number

Variable `0x024D`

On entry

A = `0xBD`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The maximum channel number for ADC conversions in the range $0 \ldots 4$

30.10.123 Osbyte 0xBE (190) ADC conversion type

Function ADC conversion type

Platform BBC Model B, BBC Model B+, Acorn Electron and BBC Master 128

Variable `0x024E`

On entry

A = `0xBE`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The ADC conversion range is always 16-bit but this variable defines the resolution within that range.

When the default 12-bit conversion is undertaken the least-significant nibble is effectively 0 as the upper 12-bits are from the conversion.

Similarly, the 8-bit conversion only uses the upper 8-bits, with the lower 8-bits effectively ignored. Due to how ADC conversions work, 8-bit conversions are about two or three times as fast as 12-bit conversions.

Other values have undefined effects.

Table 30.26: ADC Conversion Type

Value	Conversion type	Note
`0`	12-bit conversion	Default
`8`	8-bit conversion,	2–3 times faster
`12`	12-bit conversion	

30.10.124 Osbyte 0xBE (190) ADC conversion type

Function ADC conversion type

Platform BBC Master Compact

Variable `0x024E`

On entry

A = `0xBE`

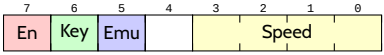
X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Setting the bits in *X* have the following effects:



bit 7 = 0 update ADVAL values from cursor keys and/or a digital joystick

bit 7 = 1 do not update ADVAL from cursor keys or digital joystick but allow updates to be handled by a sideways ROM

bit 6 = 1 Keys are entered into the keyboard buffer depending on the ADVAL(0) settings in Table 30.27.
This does not emulate in INKEY[59].

bit 5 = 0 return varying emulated numbers to ADVAL(1) *etc.*

bit 5 = 1 return fixed values to ADVAL(1) and ADVAL(2) with the values in Table 30.28

bits 3 ... 0 Emulate analogue joystick movement by returning values changing at various speeds[60].

Table 30.27: P.692

59: INKEY will detect if COPY is pressed, but it will not recognize the FIRE button on the joystick

Table 30.28: P.692

60: 7 fastest, 0 slowest

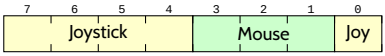


Table 30.27: Master Compact ADC Conversion Types

Bit	Function	ASCII	Key
0	Joystick Fire	0x87	COPY
1	Mouse Left	0x87	COPY
2	Mouse Middle	0x0D	RETURN
3	Mouse Right	0x7F	DELETE
4	Joystick Left	0x88	Cursor Left
5	Joystick Down	0x8A	Cursor Down
6	Joystick Up	0x8B	Cursor Up
7	Joystick Right	0x89	Cursor Right

Table 30.28: Master Compact Bit 5 return values

Direction	ADVAL(1)	ADVAL(2)
LEFT	0xFFFF	
RIGHT	0x0000	
CENTER Horizontal	0x7FFF	
CENTER	0x7FFF	0x7FFF
CENTER Vertical		0x7FFF
DOWN		0xFFFF
UP		0x0000

30.10.125 Osbyte 0xBF (191) RS-423 Use Flag

Function RS-423 Use Flag

Variable `0x024F`

On entry

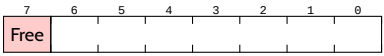
A = `0xBF`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location



If bit 7 is set then the RS-423 port is free.
Clear means it's busy.

Bits 0 ... 6 are undefined.

30.10.126 Osbyte 0xC0 (192) RS-423 6840 control register

Function RS-423 6840 control register

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Variable `0x0250`

On entry

A = `0xC0`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.127 Osbyte 0xC0 (192) RS-423 6840 control register

Function RS-423 6840 control register

Platform Acorn Electron

Variable `0x0250`

On entry

A = `0xC0`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.128 Osbyte 0xC1 (193) Read Flash Counter

Function Read Flash Counter

Variable `0x0251`

On entry

A = `0xC1`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The counter contains the number of time units until the next change between the mark and space colours.

The counter is decremented 50 times a second.

30.10.129 Osbyte 0xC2 (194) Flash Mark Colour Duration

Function Flash Mark Colour Duration

Variable `0x0252`

On entry

A = `0xC2`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.130 Osbyte 0xC3 (195) Flash Space Colour Duration

Function Flash Space Colour Duration

Variable `0x0253`

On entry

A = `0xC3`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.131 Osbyte 0xC4 (196) Keyboard Auto-repeat delay

Function Keyboard Auto-repeat delay

Variable `0x0254`

On entry

A = `0xC4`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.132 Osbyte 0xC5 (197) Keyboard Auto-repeat rate

Function Keyboard Auto-repeat rate

Variable `0x0255`

On entry

A = `0xC5`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.133 Osbyte 0xC6 (198) *EXEC File Handle

Function *EXEC File Handle

Variable `0x0256`

On entry

A = `0xC6`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.134 Osbyte 0xC7 (199) *SPOOL File Handle

Function *SPOOL File Handle

Variable `0x0257`

On entry

A = `0xC7`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.135 Osbyte 0xC8 (200) Escape and Break Effect

Function Escape and Break Effect

Variable `0x0258`

On entry

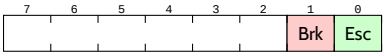
A = `0xC8`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location



This call performs two tasks.

ESCAPE

If bit 0 of X is set then the ESCAPE key will not generate an ESCAPE event, nor will the ESCAPE condition be set. This is useful when the ESCAPE key is needed for other purposes, like in games.

The ESCAPE condition can still be generated via **OSBYTE `0x7D`**[61].

61: OSBYTE `0x7D` Subsection 30.10.49

BREAK

If bit 1 of X is set then, when the BREAK key is pressed, the memory is wiped.

Example

A common practice with some games was for them to invoke this with X=3.

That then disabled ESCAPE but also wiped the memory when BREAK was pressed.

BASIC games used `*FX200,3` to do the same thing. In that case, as the memory was wiped, the basic program could not be retrieved with the **OLD** statement.

Table 30.29: Escape and Break Effect

Bit	Value	Effect
`0`	0	ESCAPE enabled
`0`	1	ESCAPE disabled
`1`	1	Memory wiped on Break

30.10.136 Osbyte 0xC9 (201) Keyboard Disable

Function Keyboard Disable

Variable `0x0259`

On entry

A = `0xC9`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

When keys are being ignored only the Break key is handled as that's tied to the processors reset line.

This is supposed to be used only by Econet and it's *REMOTE functionality.

30.10.137 Osbyte 0xCA (202) Keyboard Status

Function Keyboard Status

Variable `0x025A`

On entry

A = `0xCA`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

On the BBC Microcomputer the SHIFT enable bit 7 can be set by holding the SHIFT key down whilst the CAPS LOCK key is engaged. This enables lower-case keys to be typed when capitals are selected.

The only way to set SHIFT enable for the SHIFT LOCK key is to use this call passing $X = 144$ and $Y = 0$ or `*FX202,144`

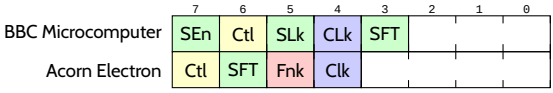


Table 30.30: Keyboard Status

bit	value	Lab	BBC Micro	Lab	Electron
3	1	SFT	SHIFT pressed		
4	0	CLk	CAPS Lock engaged	CLk	CAPS LOCK active
5	0	SLk	SHIFT LOCK engaged		
5	1			Fnk	FUNC active
6	1	Ctl	CTRL pressed	SFT	SHIFT active
7	1	SEn	SHIFT enabled	Ctl	CTRL active

30.10.138 Osbyte 0xCB (203) RS-423 Handshake Level

Function RS-423 Handshake Level

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Variable `0x025B`

On entry

A = `0xCB`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

62: Default is 9

Defines the amount of space remaining[62] in the RS-423 input buffer before input is halted by entering the full state, pulling the RTS line high.

The free space remaining is to allow for extra characters to be received before the buffer really fills up and data is lost.

The value should reflect the response time at the transmission end, time taken locally to process inbound data and the time taken for transmission between the two ends of the connection.

30.10.139 Osbyte 0xCB (203) Electron ULA IRQ mask

Function Electron ULA IRQ mask

Platform Acorn Electron

Variable `0x025B`

On entry

A = `0xCB`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The interrupt bit mask used for the Acorn Electron ULA[63].

63: Default value is `0x0C`

30.10.140 Osbyte 0xCC (204) RS-423 Suppression Flag

Function RS-423 Suppression Flag

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Variable `0x025C`

On entry

A = `0xCC`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

RS-423 input is accepted if 0, ignored otherwise. Receive errors will still cause an event.

30.10.141 Osbyte 0xCC (204) Firm Key Pointer

Function Firm Key Pointer

Platform Acorn Electron

Variable `0x025C`

Description Pointer to the currently expanding firm key

On entry

A = `0xCC`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.142 Osbyte 0xCD (205) RS-423 Selection Flag

Function RS-423 Selection Flag

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Variable `0x025D`

On entry

A = `0xCD`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

If `0x00` then data is passed to RS-423, `0x40` the cassette.

Any change takes effect only after using **OSBYTE `0x07`**[64] or **OSBYTE `0x08`**[65].

64: OSBYTE `0x07` Subsection 30.10.11
65: OSBYTE `0x08` Subsection 30.10.12

30.10.143 Osbyte 0xCD (205) Firm Key string length

Function Firm Key string length

Platform Acorn Electron

Variable `0x025D`

Description The length of the string currently being expanded from a firm key

On entry

A = `0xCD`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.144 Osbyte 0xCE (206) Econet OS call interception status

Function Econet OS call interception status

Variable `0x025E`

On entry

A = `0xCE`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

If Bit 7 is set then **OSBYTE**[66] and **OSWORD**[67] calls are routed though the Econet vector NETV.

66: OSBYTE Section 30.10
67: OSWORD Section 30.11

30.10.145 Osbyte 0xCF (207) Econet read char interception status

Function Econet read char interception status

Variable `0x025F`

On entry

A = `0xCF`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

If Bit 7 is set then **OSRDCH**[68] calls are routed though the Econet vector NETV.

68: **OSRDCH** Section 30.4

30.10.146 Osbyte 0xD0 (208) Econet write char interception status

Function Econet write char interception status

Variable `0x0260`

On entry

A = `0xD0`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

If Bit 7 is set then **OSWRCH**[69] calls are routed though the Econet vector NETV.

69: **OSWRCH** Section 30.1

30.10.147 Osbyte 0xD1 (209) Speech suppression status

Function Speech suppression status

Variable `0x0261`

On entry

A = `0xD1`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This contains the value sent to the speech processor when speech is output. `0x50` represents the SPEAK opcode and is the default value – speech enabled. Writing NOP `0x20` will disable speech.

30.10.148 Osbyte 0xD2 (210) Sound suppression status

Function Sound suppression status

Variable `0x0262`

On entry

A = `0xD2`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Sound output is disabled if this is any value other than 0.

30.10.149 Osbyte 0xD3 (211) Sound BELL channel

Function Sound BELL channel

Variable `0x0263`

Description The sound channel for the BELL Ctrl-G[70]

70: defaults to 3

On entry

A = `0xD3`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.150 Osbyte 0xD4 (212) Sound BELL amplitude/envelope

Function Sound BELL amplitude/envelope

Variable `0x0264`

Description The Amplitude or Envelope for the BELL Ctrl-G[71]

71: default is 144 `0x90` *i. e.* Amplitude -13

On entry

A = `0xD4`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.151 Osbyte 0xD5 (213) Sound BELL pitch

Function Sound BELL pitch

Variable `0x0265`

Description The frequency of the BELL Ctrl-G[72]

On entry

A = `0xD5`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

72: Defaults to 100

Errata

The New Advanced User Guide[4] shows this as 100, however the Acorn Electron Advanced User Guide[8] states 101.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.374

[8] *Adv. User Guide, Electron*, Dickens (Adder Publishing, Cambridge June 2008) P.74

30.10.152 Osbyte 0xD6 (214) Sound BELL duration

Function Sound BELL duration

Variable `0x0266`

Description Duration of the BELL Ctrl-G[73]

On entry

A = `0xD6`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

73: Defaults to 6

30.10.153 Osbyte 0xD7 (215) Startup message and Boot Options

Function Startup message and Boot Options

Variable `0x0267`

On entry

A = `0xD7`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This can only be overridden by a paged ROM on initialization or by intercepting break via the break interception vector.

If Bit 7 is set the print the OS startup message as normal. Clear then ignore the OS startup message.

Bit 0 affects how errors are handled during a `!BOOT` whilst no language has been initialized. If clear and booting from a *ROM and an error occurs then carry on but lockup if from a Disc. If set then the reverse, lockup on error in *ROM but carry on on Disc error.

Table 30.31: Boot Options

Bits		Startup	!BOOT error under Filesystem	
7	0	Message	*ROM	Disc
0	0	Ignored	Continue	Lockup
0	1	Ignored	Lockup	Continue
1	0	Print	Continue	Lockup
1	1	Print	Lockup	Continue

30.10.154 Osbyte 0xD8 (216) Soft Key Length

Function Soft Key Length

Variable `0x0268`

On entry

A = `0xD8`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The number of characters remaining in the soft key buffer for the soft key currently being expanded

30.10.155 Osbyte 0xD9 (217) Lines printed since last page pause

Function Lines printed since last page pause

Variable `0x0269`

On entry

A = `0xD9`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The number of lines printed since the last page halt. It is used to determine if output should be paused if paged mode (*e.g.* the Page Lock key) is active.

74: **OSWORD** (`0x00`) ??

OSWORD (`0x00`)[74] sets this to 0 to prevent a scroll halt during input.

30.10.156 Osbyte 0xDA (218) Number items in VDU queue

Function Number items in VDU queue

Variable `0x026A`

On entry

A = `0xDA`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This contains the number of items remaining in the VDU queue – in two's complement format.

Whilst its negative (Bit 7 set) then this is incremented for each byte received.

When it reaches 0 then the command is ready to be processed as all data has been received.

Writing 0 here is a way to abandon the VDU queue.

30.10.157 Osbyte 0xDB (219) TAB character

Function TAB character

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Variable `0x026B`

Description This is the character value returned by the TAB key – normally `0x09`.

On entry

A = `0xDB`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

It is possible to convert the TAB key into a soft key by setting this to `0x80` + *n* where n is the soft key number.

30.10.158 Osbyte 0xDB (219) External sound flag

Function External sound flag

Platform Acorn Electron

Variable `0x026B`

On entry

A = `0xDB`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Flag indicating the use of an external sound system, set by **OSBYTE `0x7E`**[75]

75: OSBYTE `0x7E` Subsection 30.10.29

30.10.159 Osbyte 0xDC (220) ESCape character

Function ESCape character

Variable `0x026C`

On entry

A = `0xDC`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The ASCII character that will generate an ESCAPE condition or event. Default is `0x1B`.

e.g. `*FX220,32` will make the Space bar the escape key, whilst `*FX220,65` the A key.

30.10.160 Osbyte 0xDD (221) Input buffer interp Cx

Function Input buffer interp Cx

Variable `0x026D`

On entry

A = `0xDD`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Defines how keys with codes `0xC0`...`CF` are handled when placed into the input buffer. Default is `0x01`.

These codes cannot be inserted from the keyboard but the OS recognizes them. RS-423 can insert these codes, as can custom keyboard routine which insert directly into the buffer.

Table 30.32: Boot Options

Value	Action
`0`	Ignore key
`1`	Expand as soft key
2 ... 255	Add value to base for ASCII code

30.10.161 Osbyte 0xDE (222) Input buffer interp Dx

Function Input buffer interp Dx

Variable `0x026E`

On entry

A = `0xDE`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Identical to **OSBYTE `0xDD`**[76] but handles keys with codes `0xD0…DF`.

76: **OSBYTE `0xDD`** P.716

30.10.162 Osbyte 0xDF (223) Input buffer interp Ex

Function Input buffer interp Ex

Variable `0x026F`

On entry

A = `0xDF`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Identical to **OSBYTE `0xDD`**[77] but handles keys with codes `0xE0…EF`.

77: **OSBYTE `0xDD`** P.716

30.10.163 Osbyte 0xE0 (224) Input buffer interp Fx

Function Input buffer interp Fx

Variable `0x0270`

On entry

A = `0xE0`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

78: **OSBYTE `0xDD`** P.716

Identical to **OSBYTE `0xDD`**[78] but handles keys with codes `0xF0`...`FF`.

30.10.164 Osbyte 0xE1 (225) Function Key status

Function Function Key status

Variable `0x0271`

On entry

A = `0xE1`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Defines how the function keys f_0 ... f_15 are interpreted when placed into the input buffer.

Table 30.32: P.716

Default is `0x01` with the options listed in Table 30.32.

30.10.165 Osbyte 0xE2 (226) Function Key + SHIFT status

Function Function Key + SHIFT status

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Variable `0x0272`

On entry

A = `0xE2`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Defines how the function keys SHIFT + $f_0 \ldots f_1 5$ are interpreted when placed into the input buffer.

Default is `0x80` with the options listed in Table 30.32.

Table 30.32: P.716

30.10.166 Osbyte 0xE2 (226) Function Key + A...P status

Function Function Key + A...P status

Platform Acorn Electron

Variable `0x0272`

On entry

A = `0xE2`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Defines how the Func + `A...P` keys are interpreted when placed into the input buffer.

Table 30.32: P.716

Default is `0x01` with the options listed in Table 30.32.

30.10.167 Osbyte 0xE3 (227) Function Key + CTRL status

Function Function Key + CTRL status

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Variable `0x0273`

On entry

A = `0xE3`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Defines how the function keys CTRL + $f_0 \ldots f_15$ are interpreted when placed into the input buffer.

Default is `0x90` with the options listed in Table 30.32.

Table 30.32: P.716

30.10.168 Osbyte 0xE3 (227) Function Key + Q...Z status

Function Function Key + Q...Z status

Platform Acorn Electron

Variable `0x0273`

On entry

A = `0xE3`

X,*Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Defines how the Func + `Q...Z` or `: ; , - . /` keys are interpreted when placed into the input buffer.

Table 30.32: P.716

Default is `0x01` with the options listed in Table 30.32.

30.10.169 Osbyte 0xE4 (228) Function Key + CTRL SHIFT status

Function Function Key + CTRL SHIFT status

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Variable `0x0274`

On entry

A = `0xE4`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Defines how the function keys CTRL + SHIFT + $f_0 \ldots f_1 5$ are interpreted when placed into the input buffer.

Default is `0x00` – no effect, with the options listed in Table 30.32.

Table 30.32: P.716

30.10.170 Osbyte 0xE5 (229) Escape action flag

Function Escape action flag

Variable `0x0275`

On entry

A = `0xE5`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

If this is 0 then the ESCAPE key has its normal action.

Any other value causes the ASCII code to be used.

30.10.171 Osbyte 0xE6 (230) Escape effects flag

Function Escape effects flag

Variable `0x0276`

On entry

A = `0xE6`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

If this is 0 then when an ESCAPE is acknowledged the following actions are performed. Any other value causes none of these to occur.

- Current EXEC file closed
- Purge all buffers[79] including the input buffer
- Reset VDU paging counter[80]
- Reset VDU queue[81]
- Clear any current soft key expansion
- Terminate any sounds being generated

79: Purge all buffers Subsection 30.10.19
80: VDU paging counter Subsection 30.10.155
81: VDU queue Subsection 30.10.156

30.10.172 Osbyte 0xE7 (231) User VIA IRQ bit mask

Function User VIA IRQ bit mask

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Variable `0x0277`

On entry

A = `0xE7`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The interrupt bit mask for the user 6522-VIA chip.

30.10.173 Osbyte 0xE7 (231) Reserved for hardware expansion

Function Reserved for hardware expansion

Platform Acorn Electron

Variable `0x0277`

On entry

A = `0xE7`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

30.10.174 Osbyte 0xE8 (232) RS-423 6850 IRQ bit mask

Function RS-423 6850 IRQ bit mask

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Variable `0x0278`

On entry

A = `0xE8`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The interrupt bit mask for the RS-423 6850 chip.

30.10.175 Osbyte 0xE8 (232) Sound semaphore

Function Sound semaphore

Platform Acorn Electron

Variable `0x0278`

On entry

A = `0xE8`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The sound semaphore

Errata

This is in the New Advanced User Guide[4] only states this is reserved for future expansion under the entries for the BBC Microcomputer.

However, the Acorn Electron Advanced User Guide[8] lists it as the Sound semaphore.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.131,247

[8] *Adv. User Guide, Electron*, Dickens (Adder Publishing, Cambridge June 2008) P.80

30.10.176 Osbyte 0xE9 (233) System VIA IRQ bit mask

Function System VIA IRQ bit mask

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Variable `0x0279`

On entry

A = `0xE9`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The interrupt bit mask for the system 6522-VIA chip.

30.10.177 Osbyte 0xE9 (233) Soft key pointer

Function Soft key pointer

Platform Acorn Electron

Variable `0x0279`

On entry

A = `0xE9`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The soft key pointer

Errata

This is in the New Advanced User Guide[4] only states this is reserved for future expansion under the entry for the BBC Microcomputer.

However, the Acorn Electron Advanced User Guide[8] lists it as the soft key pointer.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.131

[8] *Adv. User Guide, Electron*, Dickens (Adder Publishing, Cambridge June 2008) P.81

30.10.178 Osbyte 0xEA (234) Tube present flag

Function Tube present flag

Variable `0x027A`

On entry

A = `0xEA`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Set to `0x00` if the Tube is not present, or `0xFF` if it is present.

30.10.179 Osbyte 0xEB (235) Speech System present flag

Function Speech System present flag

Variable `0x027B`

On entry

A = `0xEB`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Set to `0x00` if the Speech hardware is not present, or `0xFF` if it is present.

30.10.180 Osbyte 0xEC (236) Output stream destination

Function Output stream destination

Variable `0x027C`

On entry

A = `0xEC`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The current output streams destination set by **OSBYTE `0x03`**[82]. **OSBYTE `0xEC`**[83] should only be used read it.

82: OSBYTE `0x03` Subsection 30.10.7
83: OSBYTE `0xEC` Subsection 30.10.180

30.10.181 Osbyte 0xED (237) Cursor editing status

Function Cursor editing status

Variable `0x027D`

On entry

A = `0xED`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The current cursor edit status set by **OSBYTE `0x03`**[84]. **OSBYTE `0xEC`**[85] should only be used read it.

84: OSBYTE `0x03` Subsection 30.10.8
85: OSBYTE `0xEC` Subsection 30.10.181

30.10.182 Osbyte 0xEE (238) Numeric Keypad base

Function Numeric Keypad base

Platform BBC Master 128 and BBC Master Compact

Variable `0x027E`

On entry

A = `0xEE`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The ASCII characters generated by the keys on the keypad are determined by this value.

The key code generated is done with:

$$asciiGenerated = keyAscii - 48 + numKeyPadBase$$

30.10.183 Osbyte 0xEF (239) Shadow memory use

Function Shadow memory use

Platform BBC Master 128 and BBC Master Compact

Variable `0x027F`

On entry

A = `0xEF`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Holds the value written by **OSBYTE `0x72`**[86].

86: OSBYTE `0x72` Subsection 30.10.38

0 indicates that Shadow memory should be used for all modes.

Any other value means that shadow memory is not used for the standard modes 0 ... 7.

30.10.184 Osbyte 0xF0 (240) Country Code

Function Country Code

Platform Acorn Electron, BBC Master 128 and BBC Master Compact

Variable `0x0280`

On entry

A = `0xF0`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The country the version of MOS was written for.

In most cases this is 0 for the UK. It is set to 1 for the US.

On RISC-OS this was extended with additional codes which are also listed in Table 30.33

Table 30.33: Country Code

MOS	RISC-OS	Country
0	1	UK
	2	Master
	3	Compact
	4	Italy
	5	Spain
	6	France
	7	Germany
	8	Portugal
	9	Esperanto
	10	Greece
	11	Sweden
	12	Finland
	13	Unused
	14	Denmark
	15	Norway

Table 30.33: Country Code (continued)

MOS	RISC-OS	Country
	16	Iceland
	17	Canada 1
	18	Canada 2
	19	Canada
	20	Turkey
	21	Arabic
	22	Ireland
	23	Hong Kong
	24	Russia
	25	Russia 2
	26	Israel
	27	Mexico
	28	Latin America
1	48	USA

30.10.185 Osbyte 0xF1 (241) User Flag

Function User Flag

Variable `0x0281`

On entry

A = `0xF1`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This is free for use by an application – it is not used by MOS.

The preferable method of updating or reading this flag is via **OSBYTE `0x01`**[87] but **OSBYTE `0xF1`**[88] can also be used safely.

87: OSBYTE `0x01` Subsection 30.10.5
88: OSBYTE `0xF1` Subsection 30.10.185

30.10.186 Osbyte 0xF2 (242) Serial ULA control register

Function Serial ULA control register

Platform BBC Model B, BBC Model B+, BBC Master 128 and BBC Master Compact

Variable `0x0282`

On entry

A = `0xF2`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This is a copy of the value written to the serial processor ULA control register located at address `0xFE10`, and is the only means of reading the value as the hardware is write-only. As such, **OSBYTE `0xF2`**[89] should only be used to read it.

To write then **OSBYTE `0x07`**[90] and **OSBYTE `0x08`**[91] should be used as those calls are not just safe but will work across platorms.

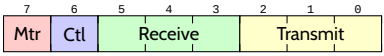
Table 30.34 on page 737 shows the values of this location pertaining to the baud rate.

The values rely on the 6850 control register being set to divide the clock rate by 64.

89: OSBYTE `0xF2`
Subsection 30.10.187

90: OSBYTE `0x07`
Subsection 30.10.11

91: OSBYTE `0x08`
Subsection 30.10.12



Bit 6 is set if the RS-423 system has control and clear if the cassette[92] has control.

Bit 7 is set if the cassette motor relay is switched on.

92: On the BBC Microcomputer the 6850 serial chip is used for the cassette

Table 30.34: Serial ULA Baud Rates

Receive 5 4 3	Transmit 2 1 0	Baud rate
0 0 0	0 0 0	19200
1 0 0	1 0 0	9600
0 1 0	0 1 0	4800
1 1 0	1 1 0	2400
0 0 1	0 0 1	1200
1 0 1	1 0 1	300
0 1 1	0 1 1	150
1 1 1	1 1 1	75

30.10.187 Osbyte 0xF2 (242) Last value written to ULA

Function Last value written to ULA

Platform Acorn Electron

Variable `0x0282`

On entry

A = `0xF2`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

On the Acorn Electron, this holds the last value written to the ULA at address `0xFE07`.

30.10.188 Osbyte 0xF3 (243) Time Clock Switch

Function Time Clock Switch

Variable `0x0283`

On entry

A = `0xF3`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

MOS maintains two copies of the internal clock where one copy can be accessed whilst the other is incremented.

This value then contains either the value 5 or 10 which points to the ram copy of the clock that is not being incremented.

Those offsets are from address `0x028D`, pointing to the clock values addresses `0x0292` and `0x0297`.

As this is updated by MOS via an interrupt this should only be read from, if used at all,by **OSBYTE `0xF3`**[93]

93: **OSBYTE `0xF3`**
Subsection 30.10.188

30.10.189 Osbyte 0xF4 (244) Soft key consistency flag

Function Soft key consistency flag

Variable `0x0284`

On entry

A = `0xF4`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This is 0 when the soft key buffer is in a consistent, stable state.

It gets set to a non-zero value during updates to or deletions from the key definitions.

If the keys are in an inconsistent state during a soft break then the soft key buffer is cleared.

30.10.190 Osbyte 0xF5 (245) Printer destination

Function Printer destination

Variable `0x0285`

On entry

A = `0xF5`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This holds the current printer destination set by **OSBYTE `0x05`**[94].

94: **OSBYTE `0x05`** Subsection 30.10.9

Writing directly to this flag does not check the previous printer setting nor notify any user printer drivers.

30.10.191 Osbyte 0xF6 (246) Printer ignore character

Function Printer ignore character

Variable `0x0286`

On entry

A = `0xF6`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This holds the current printer ignore character set by **OSBYTE `0x06`**[95].

95: **OSBYTE `0x06`** Subsection 30.10.10

30.10.192 Osbyte 0xF7 (247) Break Intercept

Function Break Intercept

Variable `0x0287`

Length 3 bytes

On entry

A = `0xF7`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

These three bytes are set to form a 6502 JMP instruction which will be during a soft break. If the first byte is not `0x4C` then this intercept code is ignored.

If configured then the code is called twice when the break occurs.

- The first time with $C = 0$ is made before the reset message is printed or the Tube initialized.
- The second time with $C = 1$ is made after the reset message and the Tube initialized.

As this is three bytes long, then **OSBYTE** s `0xF7`, `0xF8` and `0xF9` should be used to update the vector.

30.10.193 Osbyte 0xFA (250) Select memory for VDU

Function Select memory for VDU

Platform BBC Master 128 and BBC Master Compact

Variable `0x028A`

On entry

A = `0xFA`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This holds the state set by **OSBYTE `0x70`**[96].

96: **OSBYTE `0x70`** Subsection 30.10.36

30.10.194 Osbyte 0xFB (251) Select memory for display

Function Select memory for display

Platform BBC Master 128 and BBC Master Compact

Variable `0x028B`

On entry

A = `0xFB`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This holds the state set by **OSBYTE `0x71`**[97].

97: **OSBYTE `0x71`** Subsection 30.10.37

30.10.195 Osbyte 0xFC (252) Current language ROM

Function Current language ROM

Variable `0x028C`

On entry

A = `0xFC`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This holds the current language rom and is set after **OSBYTE `0x8E`**[98].

On a soft BREAK this rom will be selected and entered via it's language entry point.

On a BRK instruction, then this rom will be selected so that the current language can handle the error.

98: OSBYTE `0x8E` Subsection 30.10.68

30.10.196 Osbyte 0xFD (253) Last reset type

Function Last reset type

Variable `0x028D`

On entry

A = `0xFD`

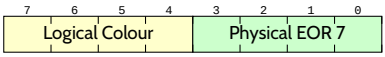
X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

This holds the type of reset last performed.



Bit 0 indicates if the machine has just been powered up.

Bit 1 indicates if the CTRL key was pressed along with the BREAK key.

Table 30.35: Last Break Type

Bits 1 0	Break type	Value
`0 1`	Power up	`1`
`0 0`	Soft Break	`0`
`1 0`	Hard Break	`2`

30.10.197 Osbyte 0xFE (254) RAM Size

Function RAM Size

Platform BBC Model B

Variable `0x028E`

On entry

A = `0xFE`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

The BBC Model A only had 16K of RAM, so it and the BBC Model B stored the amount of RAM in this location to allow MOS account for the lower memory constraints.

The value would be the page number above the last amount of RAM installed.

Table 30.36: OSByte 0xFE return

Machine	Return	RAM
BBC A	`0x40`	16K
BBC B	`0x80`	32K
BBC B+	`0x01`	
Electron	`0x00`	

30.10.198 Osbyte 0xFE (254) Reserved

Function Reserved

Platform Acorn Electron

Variable `0x028E`

On entry

A = `0xFE`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

Errata

[8] *Adv. User Guide, Electron*, Dickens (Adder Publishing, Cambridge June 2008) P.86

The Acorn Electron Advanced User Guide[8] states that this call is reserved for future Acorn expansion and that it returns 0.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.160

However, the New Advanced User Guide[4] only mentions the return value and does not mention that this is reserved.

Errata – BBC Model B+

That same page in the New Advanced User Guide is the only place where the return value of 1 for the BBC Model B+ is mentioned.

30.10.199 Osbyte 0xFE (254) Keypad Shift Key Flag

Function Keypad Shift Key Flag

Platform BBC Master 128 and BBC Master Compact

Variable `0x028E`

On entry

A = `0xFE`

X, *Y* Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

If this is set then a key on the numeric keypad is pressed along with a shift key, then the character inserted will be that of the equivalent key on the main keyboard.

e.g. SHIFT+4 would generate `$` if enabled but `4` if not enabled.

30.10.200 Osbyte 0xFF (255) Startup Options

Function Startup Options

Variable `0x028F`

On entry

A = `0xFF`

X, Y Refer to Subsection 30.10.1

On exit

X Old value

Y Value of next location

When this location is changed with **OSBYTE `0xFF`**[99] then it's state takes effect at the next soft reset. However, a hard reset will reset it to its hardware configured value.

99: OSBYTE `0xFF` Subsection 30.10.200

Table 30.37 shows the bits for this option

Table 30.37: P.750

	7	6	5 4	3	2 1 0
BBC Model B	FS		Timings	Boot	Screen Mode
Acorn Electron		PC	Timings	Boot	Screen Mode
BBC Master 128				Boot	

BBC Model B

On the BBC Model B there are a set of option switches.

On type 1 keyboards (the most common) they are located at the bottom right of the keyboard next to the space bar.

On type 2 keyboards they are on the right and oriented vertically.

The links column in Table 30.37 shows these links, which are numbered in reverse order to the associated bit number.

Table 30.37: P.750

Table 30.38 shows the values for the drive timing bits.

Table 30.38: P.750

Acorn Electron

The Acorn Electron has no switches so this is initially set to `0xFF` on power up. This means that the initial screen mode is 7 – however, as the Acorn Electron has no Mode 7 it defaults to Mode 6.

Table 30.39: P.750

Table 30.39 shows the values for the drive timing bits. Those and the pre-compensation bit are used for the Plus 3 expansion if fitted.

Any changes will only take effect after the next soft reset.

Errata

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.421

[8] *Adv. User Guide, Electron*, Dickens (Adder Publishing, Cambridge June 2008) P.87

The New Advanced User Guide[4] mentions this is for the BBC Model B and it's keyboard links.

The Acorn Electron Advanced User Guide[8] concurs along with the differences with the platform.

However, they differ in several respects. From The New Advanced User Guide:

Bit 7 listed for DFS and NFS but no mention that this is unused on the Acorn Electron

Bit 6 listed as '*not used by OS (reserved for future applications)*' except that this is the drive pre-compensation flag on the Acorn Electron

Bits 4,5 show the BBC Microcomputer drive timings in Table 30.38. However, the Acorn Electron Advabced User Guide lists different timings, reproduced here in Table 30.39. Although they look similar to the BBC Model B 1770 step-timings, the bits associated with them are complemented.

Errata – BBC Master 128

The New Advanced User Guide[4] does not mention the BBC Master 128 or by name, only referencing the BBC Microcomputer and Acorn Electron. This I can understand as the configuration for how the BBC Master 128 and BBC Master Compact starts is stored in CMOS RAM or EEPROM respectively so this call is redundant.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.421

However, the BBC Master 128 Reference Manual Part One[9] does list this call stating that the "call provides a means of reading and interchanging the effects of Break and Shift+Break".

[9] *Master Series Ref. Manual Part 1*, (Jan. 1986) P.D.2-84

- ***FX255,0,247** interchanges the effects
- ***FX255,8,247** restores normal functions

This is unexpected as this functionality is configured with the ***CONFIGURE BOOT**[100] command, and stored in CMOS[101].

100: ***CONFIGURE BOOT** Section 28.4
101: **Table D.13:** P.993 Byte 10, Bit 4

As the example also used 247 for Y, *e.g.* the complement of 8, the other bits are either being preserved or have some internal function that is undocumented[102]

102: Could this be configured from CMOS on power-up and then MOS then uses this variable?

In either case, the only way this setting may be of use is to change how subsequent soft boots work, just like on the Acorn Electron, without having to change CMOS which would persist after the machine was turned off.

Errata – BBC Master Compact

I do not have a BBC Master Compact or documentation specific to it, and it is not mentioned anywhere for this call, but I'm presuming this will be identical to the BBC Master 128.

I'm only mentioning this here as this call could be beneficial on the BBC Master Compact compared to ***CONFIGURE** as it uses EEPROM which has a limited number of write cycles.

Table 30.37: Startup Options for OSBYTE `0xFF`

Link	Bit	State	Bit set	Bit clear	BBC B	BBC B+	Electron	Master	Compact
1	7	Default Filesystem	DFS	NFS	⋆	⋆			
	6	Drive pre-compensation	Required	None			⋆		
3, 4	5,4	Drive timings			⋆	⋆	⋆		
5	3	Boot mode	SHIFT+BREAK	BREAK	⋆	⋆	⋆	⋆	⋆
6,7,8	2,1,0	Screen mode			⋆	⋆	⋆		

Table 30.38: Startup Drive timings - BBC Model B

Bits	8271 Timing *ms*		Head	1770 Timing *ms*		1772 Timing *ms*	
5 4	Step	Settle	Load	Step	Settle	Step	Settle
0 0	4	16	0	6	30	6	15
0 1	6	16	0	12	30	12	15
1 0	6	50	32	20	30	2	15
1 1	24	20	64	30	30	3	15

Table 30.39: Startup Drive timings - Acorn Electron

Bits	Speed
5 4	*ms*
0 0	30
0 1	20
1 0	12
1 1	6

30.11 OSWORD - OS functions with a parameter block

Function	OSWORD
Address	FFF1
Vector	020C

OSBYTE calls are a powerful and flexible way of invoking many of the available operating system facilities.

They are used for functionality which only requires a simple one or two byte parameter to the function and similarly for the return.

Simpler actions are implemented via the **OSBYTE**[103] call which takes parameters with just the X and Y registers.

103: **OSBYTE** Section 30.10

Table D.4 lists the documented **OSWORD**. Unlike **OSBYTE**, this includes not only the standard MOS **OSWORD** calls, but those from third party software, when known.

Table D.4: P.973

30.11.1 Parameters

OSWORD on the 6502uses the Accumulator (A register) to define the action and the X and Y registers as a pointer to a parameter block. The size of this block of memory is specific to the action.

Other processors attached via the Tube use different registers. *e.g.* on the Z80, the A and HL registers are used.

In the definitions for each call later in the chapter, each parameter block is listed with a minimum of three columns: Offset, the byte offset from the address pointed to by XY, Length the number of bytes the parameter takes and what the parameter is.

At the end of each table is the size of memory the block requires. This is important as some sources of information either do not show that value, or it isn't obvious what the true size is.



Take a look at the errata for **OSWORD** `0x0E` in Subsection 30.11.20 on why I show the parameter block size this as it would be pretty easy to cause a buffer overrun depending on what documentation you were using.

30.11.2 MOS OSWORD calls

OSWORD calls with values `0x00` ... `0x0F` are defined by MOS. Most of these calls are available on every machine, although some of the higher call numbers are only present from the BBC Master 128. The system columns in Table D.4 show which systems a call is supported under.

Table D.4: P.973

30.11.3 Application and Filesystem calls

OSWORD calls with values `0x10` ... `0xDF` are available for use by Paged ROMs and filesystems like DFS and ADFS.

Some calls within this range are used by multiple filesystems, *e.g.* **OSWORD `0x7F`**[104] issues a command to the floppy disk controller.

104: **OSWORD `0x7F`** Subsection 30.11.22

30.11.4 Unknown OSWORD calls

Any unknown **OSWORD** calls with values `0xE0` ... `0xFF` are passed to the user vector[105]. All other unknown calls as passed to the Paged ROMs using a Paged ROM service call with A=`0x08`.

105: **USERV** `0x0200`

The A, X and Y registers at the time of the call are stored in zero-pages[106] addresses `0xEF`, `0xF0` and `0xF1` respectively.

106: **Table D.5:** P.975

Unknown OSWORD calls and DFS 0.90 Bug

Apparently there is a bug with BBC Model B[4] machines running DFS 0.90 where it treats any unknown **OSWORD** call in the `0xE0` ... `0xFF` range as if they were `0x7F` so Paged ROMs cannot intercept them.

This does not happen for later versions of DFS or on any other machine.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.111

30.11.5 OSWord 0x00 (0) Read line from current input stream

Function Read line from current input stream

On entry

A = `0x00`

XY Parameter block address

On exit Escape condition

Carry = 0 Carriage return terminated input

Carry = 1 Escape condition terminated input

Y Line length including carriage return if used

This call provides a common means to request input from the user where that input needs to be terminated by a carriage return. This is used for command line input, e.g. in BBC Basic when entering a program, BASICs **INPUT** statement or the ***CLI**[107] command.

The call takes a 5 byte parameter block,[108] where the first two bytes hold the address where the input should be placed, followed by the maximum length of the line.

This target buffer for the input must be that length +1 to include the carriage return string terminator[109].

Offset *Hex*	Length	Parameter
0 00	2	Buffer address for input
2 02	1	Maximum length of line
3 03	1	Min accepted character value
4 04	1	Max accepted character value
5 05		Block size

107: ***CLI** Section 25.3 part of Lancaster assembler

108: **Table 30.40:** P.753

109: CR is present if the input terminated with a CR. It's not terminated if Escape was used

Table 30.40: OSWord 00 parameter block

Editing the input

The Delete key will delete the previous character, whilst Control-U `0x15` deletes the entire line.

Terminating input

The input line will terminate with either a Carriage Return or the Escape character, indicated by the state of the Carry flag when the call returns.

If carry is set then an Escape occurred, and it must be acknowledged with **OSBYTE `0x7E`**[110].

110: **OSBYTE `0x7E`**
Subsection 30.10.50

Line limit

The maximum length of the input is defined in byte 2 of the parameter block[111]. If the line limit is reached then extra characters are ignored and a bell `0x07` is output.

111: **Table 30.40:** P.753

Restricting acceptable characters

Bytes 3 and 4 of the parameter block limit which characters can be used in the input. Any character not within this inclusive-range will not be accepted.

e.g. using `0x40` and `0x49` you can limit the input to just numeric digits. `0x30` and `0x7E` means the normal ASCII characters can be used.

30.11.6 OSWord 0x01 (1) Read system clock

Function Read system clock

On entry

A = `0x01`

XY Parameter block address



In BBC Basic, the TIME variable uses **OSWORD `0x01`** when reading and **OSWORD `0x02`** when writing the variable

On all Acorn MOS based machines, clocks are 5 bytes long and are updated once every centisecond. This call reads from the system clock which records time and is used by the BBC Basic TIME function.

The parameter block shown in Table 30.41 is a 40-bit, 5-byte counter where byte 0 is the least significant byte of the counter and byte 4 the most significant byte.

Table 30.41: MOS Clock

Offset *Hex*	Length	Parameter
0 00	5	40-bit clock value
5 05		Block size

30.11.7 OSWord 0x02 (2) Write system clock

Function Write system clock

On entry

A = `0x02`

XY Parameter block address

This call complements **OSWORD `0x01`**[112] allowing for the system clock to be reset to a specific time.

112: **OSWORD `0x01`** Subsection 30.11.6

30.11.8 OSWord 0x03 (3) Read interval timer

Function Read interval timer

On entry

A = `0x03`

XY Parameter block address

The interval timer is the second clock maintained by MOS. Like the system clock, it is incremented once every centisecond. However, when it crosses zero it generates an event which software can use to perform some action.

30.11.9 OSWord 0x04 (4) Write interval timer

Function Write interval timer

On entry

A = `0x04`

XY Parameter block address

This call complements **OSWORD `0x03`**[113] allowing for the interval timer to be reset.

As the interval timer generates an event when it crosses zero, this needs to be set to a negative value otherwise you would be waiting a very long time[114] for the event to be generated.

113: **OSWORD `0x03`** Subsection 30.11.8

114: 2^{40} centiseconds is about 348 years

30.11.10 OSWord 0x05 (5) Read byte of I/O processor memory

Function Read byte of I/O processor memory

Description Read a byte of memory from the I/O processor

On entry

A = `0x05`

XY Parameter block address

This call reads a byte of memory from the I/O processor. It is intended for use by software to access memory over the Tube from a Second Processor.

The parameter block[115] is 5 bytes long. The first 4 bytes hold the address in the I/O processor whilst the last byte holds the value of the byte after the call.

115: **Table 30.42:** P.757

To enable filesystems to recognize which processor's memory a file references, the upper 2 bytes of the address are `0xFFFF` for the I/O processor.

e.g. DFS uses an 18-bit address for the load and execute addresses in the file catalogue with bits 16 and 17 set to 1 to indicate the I/O processor. If both are set then DFS OR's the main address with `0xFFFF0000`.

Table 30.42: Read/Write I/O Processor memory parameter block

Offset *Hex*	Length	Parameter
0 00	4	32-bit Address
4 04	1	Byte to write or read
5 05		Block size

BBC Master 128

In normal use this call reads from the main memory. However, if the top two bytes are `0xFFFE` then it will read from sideways RAM.

BBC Model B+

In normal use this call reads from the main memory. However, if the top two bytes are `0xFFFE` then it will read from the 12K of sideways RAM.

Paged ROM

On some systems the prefix `0xFFrr` is used to reference a Paged ROM – where rr is the ROM number.

Errata

The BBC WikiOsword05BeebWikiMDFS mentions both the `0xFFFE` and `0xFFrr` prefixes as being on "Some systems", but it does not mention which ones.

The New Advanced User Guide[4] mentions only `0xFFFE` as being for the BBC Model B+ only. This too is inaccurate as the BBC Microcomputer Master Series Reference Manual Part 1 mentions it stating that `0xFFFE` can be used to read from the sideways RAM[9], without mentioning the 12K limitation.

Neither books mention the `0xFFrr` at all.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.347

[9] *Master Series Ref. Manual Part 1*, (Jan. 1986) P.D.3-10

30.11.11 OSWord 0x06 (6) Write byte of I/O processor memory

Function Write byte of I/O processor memory

On entry

A = `0x06`

XY Parameter block address

This call complements **OSWORD `0x05`**[116] allowing for a byte to be written to the I/O processor. The byte written is in byte 4 of the parameter block[117].

The address follows the same rules as for **OSWORD `0x05`**

116: OSWORD `0x05` Subsection 30.11.10

117: Table 30.42: P.757

30.11.12 OSWord 0x07 (7) Perform a SOUND command

Function Perform a SOUND command

On entry

A = `0x07`

XY Parameter block address

This takes an 8 byte parameter block[118] consisting of 4 2-byte parameters. The BBC Basic SOUND command uses this call.

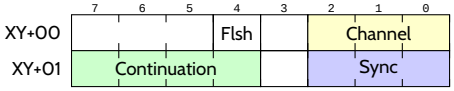
118: **Table 30.43:** P.759

Table 30.43: OSWord SOUND parameter block

Offset *Hex*	Length	Parameter
0 00	2	Channel number
2 02	2	Amplitude
4 04	2	Pitch
6 06	2	Duration
8 08		Block size

Channel number

The following bits relate to the Channel field in Table 30.43.



Channel Channel Number 0...3

Flsh Flush control

0 queue the sound

1 Terminate any current sound and start this one

Sync Synchronisation – 0...3

0 play when queue is ready, *e.g.* empty

1...3 delay sound until this number of sound channels have sound commands with a similar Sync number ; 0.
e.g. this synchronizes sounds, so they start together.

Continuation How the sound behaves with anything already playing

0 sound should be produced when the channel is ready

1 the preceding sound in this channel should finish its decay phase of the envelope. The sound with this set is not produced at all.

> 1 Passed to Paged ROMs as an unrecognized sound command

Amplitude

Signed 16-bit integer but should be in the range −15 ... 4 `0xFFF0` ...`0x0004`.



If both the cassette and RS-423 interfaces are not in use then envelope numbers 5 ... 16 are also available

Values in range 0 ··· − 15 specify a fixed amplitude for the sound, with 0 quietest, -15 loudest.

Values in the range 1 ... 4 select an envelope.

Pitch

Value between 0 ... 255.

For tone channels 1 ... 3 it specifies the pitch[119] of the note in intervals of a quarter of a semitone. Middle C has a value of 52.

119: Pitch can be modulated by a pitch envelope

For noise channel 0 it specifies the tonal quality of the noise, shown in Table 30.44.

Table 30.44: OSWord SOUND pitch on noise channel

pitch	Frequency	Type	Note
0	High	Periodic noise	Periodic noise
1	Medium	Periodic noise	Periodic noise
2	Low	Periodic noise	Periodic noise
3		Periodic noise	Periodic noise
4	High	frequency white noise	frequency white noise
5	Medium	White noise	White noise
6	Low	White noise	White noise
7		White noise	White noise

Duration

Value between 0 ... 254 representing the duration

30.11.13 OSWord 0x08 (8) Define an ENVELOPE

Function Define an ENVELOPE

On entry

A = `0x08`

XY Parameter block address

This takes a 14 byte parameter block[120] defining the envelope[10].

The BBC Basic ENVELOPE command uses this call.

120: **Table 30.45:** P.763

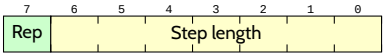
[10] *BBC Micro. User Guide*, (Oct. 1984) P.217

Envelope Number

Value between 0 ... 4 defining the envelope.

If both cassette and RS-423 are not in use then this can go up to 16 `0x10` as those extra envelopes are stored in the input buffer used by those two interfaces.

Step length



The number 0 ... 127 of centiseconds used for each step in the envelope definition.

Bit 7 if set marks the envelope not to repeat. If the sound lasts longer than the envelope then the pitch setting at the end of the envelope will be maintained until the sound finishes.

If bit 7 is cleared, then the envelope is repeated as many times as necessary until the sound completes.

Pitch change rates

These parameters, in the range −128 ... 127, define the rate at which the pitch[121] should change per step length. There are three of these parameters, one each for each section of the envelope.

121: in pitch units, $\frac{1}{4}$ *semitone*

Pitch length

The length of each section of the envelope, in units of step length. There are three of these parameters, one each for each section of the envelope.

Amplitude change rate

These parameters, in the range −128 ... 127 define the rate amplitude changes per step length during a specific phase.

There are four of these, one each for the attack, decay, sustain and release phases.

The units here is $\frac{1}{8}$ that of the amplitude units used in **OSWORD `0x07`**.

Attack target level

The amplitude that must be reached to end the attack phase. In the range 0 ... 126.

Decay target level

The amplitude that must be reached to end the decay phase. In the range 0 ... 126.

Table 30.45: OSWord ENVELOPE parameter block

Offset	*Hex*	Length	Parameter	Min	Max
0	00	1	Envelope number	1	4
1	01	1	Step duration in centiseconds	0	127
2	02	1	Pitch change per step in section 1	-128	127
3	03	1	Pitch change per step in section 2	-128	127
4	04	1	Pitch change per step in section 3	-128	127
5	05	1	Number of steps in section 1	0	255
6	06	1	Number of steps in section 2	0	255
7	07	1	Number of steps in section 3	0	255
8	08	1	Amplitude change per step during attack	-127	127
9	09	1	Amplitude change per step during decay	-127	127
10	0A	1	Amplitude change per step during sustain	-127	127
11	0B	1	Amplitude change per step during release	-127	0
12	0C	1	Target level at end of attack	0	126
13	0D	1	Target level at end of decay	0	126
14	0E		Block size		

30.11.14 OSWord 0x09 (9) Read pixel value

Function Read pixel value

On entry

A = `0x09`

XY Parameter block address

This call returns the status of a screen pixel at the supplied coordinates in a 5 byte parameter block[122]. The logical colour of the pixel is written to byte 4.

122: **Table 30.46:** P.764

If the pixel was off the screen then byte 4 is set to `0xFF`.

Table 30.46: OSWord Read pixel value parameter block

Offset *Hex*	Length	Parameter
0 00	2	X co-ordinate
2 02	2	Y co-ordinate
4 04	1	Logical colour
5 05		Block size

Notes

1. If the graphics origin has been redefined using VDU 27 then these coordinates should be relative to that new origin.
2. This call must not be used in an interrupt or event service routine.

30.11.15 OSWord 0x0A (10) Read character definition

Function Read character definition

On entry

A = `0x0A`

XY Parameter block address

This call takes a character ASCII code in byte 1 of a 9 byte parameter block[123], returning the characters definition in the following 8 bytes.

123: **Table 30.47:** P.765

Table 30.47: OSWord Read character definition

Offset *Hex*	Length	Parameter
0 00	1	Character required
1 01	8	Character definition
9 09		Block size

Notes

1. The characters with ASCII codes 0 ... 31 do not have a character definition.
2. Character 127 does have a definition, but it isn't usable via VDU.
3. This call must not be used in an interrupt or event service routine.

30.11.16 OSWord 0x0B (11) Read palette for given logical colour

Function Read palette for given logical colour

On entry

A = `0x0B`

XY Parameter block address

This call takes a logical colour in byte 1 of a 5 byte parameter block[124], the physical colour in byte 1. Bytes 2, 3 and 4 are 0 for future expansion.

124: **Table 30.48:** P.766

Table 30.48: OSWord Read palette

Offset *Hex*	Length	Parameter
0 00	1	Logical colour
1 01	1	Physical colour
2 02	3	0 Reserved for future use
5 05		Block size

30.11.17 OSWord 0x0C (12) Write palette for given logical colour

Function Write palette for given logical colour

On entry

A = `0x0C`

XY Parameter block address

This call complements **OSWORD `0x0B`**[125] and writes the specified palette. It takes the same parameter block as that call.

125: **OSWORD `0x0B`** Subsection 30.11.16

You can also issue this **OSWORD** call with VDU 19 followed by each of the 5 bytes.

30.11.18 OSWord 0x0D (13) Read previous and current graphics cursor position

Function Read previous and current graphics cursor position

On entry

A = `0x0D`

XY Parameter block address

The last two coordinates are kept to allow for operations like filling triangles.

This call will store those two coordinates in a 8 byte parameter block[126].

126: **Table 30.49:** P.767

Table 30.49: OSWord Read palette

Offset *Hex*	Length	Parameter
0 00	2	Previous X co-ordinate
2 02	2	Previous Y co-ordinate
4 04	2	Current X co-ordinate
6 06	2	Current Y co-ordinate
8 08		Block size

30.11.19 OSWord 0x0E (14) Read real time clock

Function Read real time clock

Platform BBC Master 128

On entry

A = `0x0E`

XY Parameter block address

When this call is made, the first byte can have one of three values which determine what result is returned.

(XY)=0)

If the first byte is 0 then it takes a 25 byte parameter block[127].

127: **Table 30.50:** P.771

This block then receives a string representing the time from the RTC, with the format:

"`ddd,nn mmm yyyy.hh:mm:ss`"

This string will be terminated with a carriage return `0x0D`.

(XY)=1)

If the first byte is 1 then it takes a 7 byte parameter block[128].

128: **Table 30.51:** P.771

On return, the block will contain the time in BCD.

(XY)=2)

If the first byte is 2 then it takes a 25 byte parameter block.

On entry, bytes 1 to 7 contain the BCD time as per Table 30.51. Note that in this call it starts from byte 1 and not 0.

On return, the block will contain the time converted to a 25 byte string as per Table 30.50.

Errata

Not really an error, but a warning of a possible buffer overrun.

The New Advanced User Guide[4] states that this returns a 24-byte character string, but at the end of the table describing the format it mentions a carriage return follows. This makes the returned string 25 bytes long in total, but it does not state the actual size.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.354

The BBC Microcomputer Master Series Reference Manual Part 1 confirms that the parameter block is 25 bytes[9] long.

[9] *Master Series Ref. Manual Part 1*, (Jan. 1986) P.D.3-23

Be aware of this if you use a 24 byte parameter block because you read that but did not notice the `0x0D` string terminator. If you didn't notice then the byte after the block would be overwritten with `0x0D`.

This is the reason why in this chapter, each parameter block shown includes the block size at the end. In addition, the offset column is generated and not typed, to ensure that it too is accurate.

30.11.20 OSWord 0x0E (14) Read real time clock

Function Read real time clock

Platform BBC Master Compact

On entry

A = `0x0E`

XY Parameter block address

The BBC Master Compact does not have a realtime clock so this just returns a static string:

"`Fri,31 Dec 1999.23:59:59`"

Errata

The New Advanced User Guide[4] only mentions it returns the static time. It does not mention what happens with the call is used on that machine – I'm presuming it just processes the static string.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.353

Table 30.50: OSWord RTC Clock Parameters

Offset	*Hex*	Length	Parameter	Format
0	00	3	Day of week, *e.g.* Mon, Tue	ddd
3	03	1	*Comma separator*	,
4	04	2	Day of month	nn
6	06	1	*Space separator*	␣
7	07	3	3 character month, *e.g.* May	mmm
10	0A	1	*Space separator*	␣
11	0B	4	The year, *e.g.* 1988	yyyy
15	0F	1	*Period separator*	.
16	10	2	Hour in 24hr notation	hh
18	12	1	*Colon separator*	:
19	13	2	Minutes	mm
21	15	1	*Colon separator*	:
22	16	2	Seconds	ss
24	18	1	*Carriage Return terminator*	`0x0D`
25	19		Block size	

Table 30.51: OSWord RTC Clock BCD Parameters

Offset	*Hex*	Length	Parameter	Min	Max
0	00	1	Year	`00`	`99`
1	01	1	Month	`01`	`12`
2	02	1	Day of month	`01`	`31`
3	03	1	Day of week, Sun...Sat	`01`	`07`
4	04	1	Hours	`00`	`23`
5	05	1	Minutes	`00`	`59`
6	06	1	Seconds	`00`	`59`
7	07		Block size		

30.11.21 OSWord 0x0F (15) Write real time clock

Function Write real time clock

Platform BBC Master 128

On entry

A = `0x0F`

XY Parameter block address

There are three actions available for this call, all based in the first byte of the parameter block.

Change the time only

If the first byte is 8, then this takes the parameter block in Table 30.52.

Table 30.52: P.773

Change the date only

If the first byte is 15, then this takes the parameter block in Table 30.53.

Table 30.53: P.773

Change the date and time

If the first byte is 25, then this takes the parameter block in Table 30.54.

Table 30.54: P.774

Errata

The New Advanced User Guide[4] does not mention that the BBC Master Compact doesn't have a realtime clock in the writeup for this call, so there is nothing to say what would happen if this call was made on that machine.

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.355

Table 30.52: OSWord RTC Clock set new time parameters

Offset	*Hex*	Length	Parameter	Format	Min	Max
0	00	1	Block type	`0x08`		
1	01	2	Hours	hh	`00`	`23`
3	03	1	ASCII 58	:		
4	04	2	Minutes	mm	`00`	`59`
6	06	1	ASCII 58	:		
7	07	2	Seconds	ss	`00`	`59`
9	09		Block size			

Table 30.53: OSWord RTC Clock set new date parameters

Offset	*Hex*	Length	Parameter		Format	Min	Max
0	00	1	Block type		`0x0F`		
1	01	3	Day of week	Sun...Sat	ddd	`01`	`07`
4	04	1	ASCII *44 comma*		`,`		
5	05	2	Day of month		nn	`01`	`31`
7	07	1	*Space*		␣		
8	08	3	Month	Feb	mmm		
11	0B	1	*Space*		␣		
12	0C	4	Year	1995	yyyy		
16	10		Block size				

30.11.22 OSWord 0xFF (255) Read/Write I/O processor memory Z80

Function Read/Write I/O processor memory Z80

On entry

A = `0xFF`

XY Parameter block address

This **OSWORD** command is only available from the Z80 second processor. It can read and write blocks of memory

Table 30.54: OSWord RTC Clock set new date and time parameters

Offset	*Hex*	Length	Parameter		Format	Min	Max
0	00	1	Block type		`0x18`		
1	01	3	Day of week	Sun…Sat	ddd	`01`	`07`
4	04	1	ASCII *44 comma*		,		
5	05	2	Day of month		nn	`01`	`31`
7	07	1	*Space*		␣		
8	08	3	Month	Feb	mmm		
11	0B	1	*Space*		␣		
12	0C	4	Year	1995	yyyy		
16	10	1	ASCII *46 period*		.		
17	11	2	Hours		hh	`00`	`23`
19	13	1	ASCII *58*		:		
20	14	2	Minutes		mm	`00`	`59`
22	16	1	ASCII *58*		:		
23	17	2	Seconds		ss	`00`	`59`
25	19		Block size				

from the I/O processor. It is not available on any other second processor as it's part of Z80-OS.

Operation type

This byte defines the operation. 0 = read, 1 = write

Table 30.55: OSWord Z80 parameters

Offset	*Hex*	Length	Parameter	Format
0	00	1	`0x0D`	
1	01	1	`0x01`	
2	02	4	I/O processor address	
6	06	4	Z80 processor address	
10	0A	2	Number bytes to transfer	
12	0C	1	Operation type	
13	0D		Block size	

Page intentionally blank

Operating System *Commands 31

The **OSCLI**[1] API and the simple CLI available when no language is in use provides access to commands provided by MOS, the current filesystem and Paged ROMs.

Many languages, including BBC Basic, and some applications also support these comands by exposing them with the "*" character[2]. Hence, in BBC Basic you can execute any OS command by starting it with "*".

The following list is those provided by MOS. It does not include those from filesystems or Paged ROMs. Those are listed elsewhere in this book, where appropriate.

1: **OSCLI** Section 30.7 on page 600

2: *is not needed with the OSCLI interface but if present is ignored

31.1 *CODE x,y (*CO.) Execute User Code

The ***CODE** command executes machine code pointed to by the **USERV** vector. It takes two optional integer parameters, each can take a value between 0 ... 255 which are passed to the routine in the X and Y registers. The Accumulator (A register) will be 0 when **USERV** is invoked.

31.2 *FX a,x,y Perform OSBYTE call

Function Perform **OSBYTE** call

Syntax ***FX** A [, *X* [, *Y*]]

The ***FX** command allows for any **OSBYTE** call to be performed from a command line.

The command takes either one, two or three numeric parameters separated by a comma. Those parameters

map to the A, X and Y registers when performing the underlying **OSBYTE** call.

The X and Y parameters are optional, and will be treated as 0 if absent. However, if X is absent then Y is also required to be absent.

The **OSBYTE** calls are described in full in Section 30.10.

31.3 *LINE text (*LI.) Execute User Code

The ***LINE** command executes machine code pointed to by the **USERV** vector. The Accumulator (A register) will be 1 when **USERV** is invoked with the X and Y registers pointing to the text of the command. The text will be terminated with `0x0D`.

31.4 *MOTOR n (*M.) Turn cassette motor relay on or off

The ***MOTOR** command turns on or off the cassette motor relay.

If $n = 0$ then the relay is open so the cassette motor is turned off. If $n = 1$ then the relay is closed, turning on the cassette motor.

This is the same as **OSBYTE 0x89**[3]

3: **OSBYTE 0x89** Subsection 30.10.63 on page 647

31.5 *TAPE n (*T.) Select tape filesystem

The ***TAPE** command selects the TAPE[4] filesystem allowing access to software stored on cassettes.

4: TAPE Chapter 33 on page 787

If *n* is not provided then it selects the default baud rate of 1200 Table 30.19 on page 650 lists the available values of n.

On the Acorn Electron this only switches filesystem as it only supports 1200 baud.

This is the same as **OSBYTE `0x8C`**[5]

5: **OSBYTE `0x8C`** Subsection 30.10.66 on page 650

31.6 *TV s,i
Configure TV output

The **`*TV`** command configures the TV output from the BBC Microcomputer. It is not supported on the Acorn Electron.

It adjusts the vertical offset of the image by either moving the picture up or down the specified number of lines. The first parameter defines the number if lines the screen is moved. 1 would move it up one line, 2 two lines *etc.*. In the opposite direction, 255 would move it down one line, 254 two.

The TV's interlace option can also be turned on or off with this command. If the second parameter is 0 then interlace is turned on. 1 turns it off.

This is the same as **OSBYTE `0x90`**[6]

6: **OSBYTE `0x90`** Subsection 30.10.71 on page 652

31.7 Media

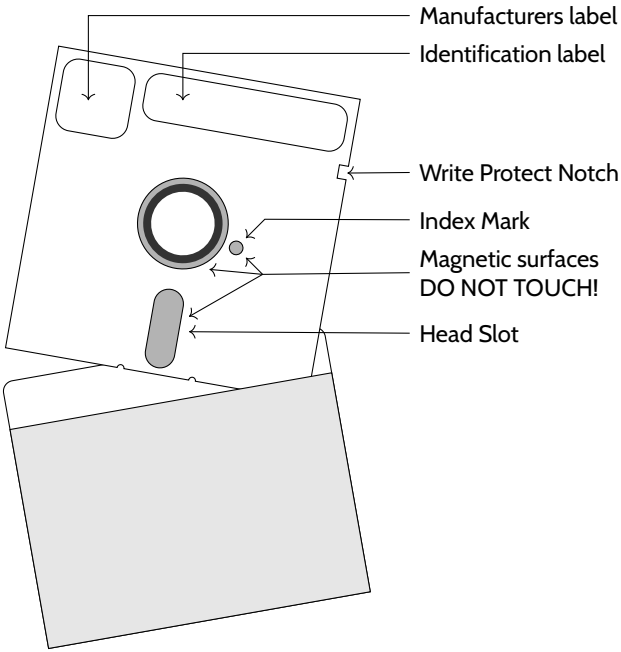


Figure 31.1: Anatomy of a $5\frac{1}{4}$" floppy disc

Filesystems

Table of contents

32 Commands common to all filesystems **783**

33 Cassette Filesystem **787**

34 ROM Filesystem **789**

35 Acorn DFS **791**
35.1 Commands 791
35.2 DFS Internals 800

36 Acorn ADFS **811**

37 LanManager File System **813**
37.1 About the SPROW 10/100 Network interface 813
37.2 Networking Commands 814
37.3 CIFS Commands 820

38 Network File System **825**

39 Alternative File Systems **827**
39.1 Acorn System DFS 827
39.2 Angus Duggan HDFS 827
39.3 Opus DDOS 830
39.4 Watford DFS 831
39.5 Watford DDFS 833
39.6 Watford 62 Disk File System 833
39.7 Telesoft File System 834
39.8 IEEE File System 834

Commands common to all filesystems

32

The commands in this section are commands that relate to the current file system. Some are provided by the operating system so are available to all file systems, others by specific filesystems.

Table E.1 on page 998 lists the common commands and which machines they are available under.

32.0.1 *CAT

Description Display the filesystem catalogue

Syntax ***CAT**

short ***.**

Example *CAT 0
CAT
.

See also Acorn DFS[1]

1: Acorn DFS ??

This command is used to show the files present on a disc or cassette.

32.0.2 *EXEC

Description Read a file and pass to an application as if the content was typed on the keyboard.

See also *BUILD[2]

2: *BUILD Subsection 35.1.3 on page 792

This command is provided by the Operating System so is available to all file systems.

32.0.3 *LOAD

Description Read a file from disk into memory.

Syntax ***LOAD** filename

***LOAD** filename address

This command will load a file into memory. If the address is not specified, then this will use the load address of the file when it was saved.

To support second processors over the Tube interface, if the address begins with 0xFFFFxxxx then it refers to the I/O processor and it's memory, whilst any other address refers to the second processor's memory.

The number of bit's available to store this address depends on the individual file system. *e.g.* for Acorn DFS, it supports 18-bit addresses, but if both bits 16 and 17 are set then it will or the address with 0xFFFF0000 to get the I/O processor's address.

32.0.4 *OPT

Description Pass options to current file system

Syntax ***OPT** X,Y

***OPT** X

***FX** 139,X,Y

***FX** 139,X

See also Table 32.1 on the facing page

This command passes an option to the current file system. The meaning of the values passed are dependent upon the individual file system.

Note: `*OPT` is the same as `*FX139` which calls the **??** on page ??vol3:mos:osbyte:hex:8B MOS call.

Table Table 32.1 shows links to sections elsewhere which define the parameters specific to a file system.

Table 32.1: Acorn DFS *OPT4 parameters

Filesystem	*OPT	Action	Section
Acorn DFS	*OPT 1,n	Enable tracing	Subsection 35.1.18 on page 798
	*Opt 4,n	Set boot options	Subsection 35.1.19 on page 798

32.0.5 *RUN

Description Load and execute a machine code program from disc.

This command is provided by the Operating System so is available to all file systems.

32.0.6 *SAVE

Description Save a block of memory to a file on disc.

Syntax *SAVE filename

*SAVE filename address

This command will load a file into memory. If the address is not specified, then this will use the load address of the file when it was saved.

To support second processors over the Tube interface, if the address begins with 0xFFFFxxxx then it refers to the I/O processor and it's memory, whilst any other address refers to the second processor's memory.

The number of bit's available to store this address depends on the individual file system. *e.g.* for Acorn DFS, it supports 18-bit addresses, but if both bits 16 and 17 are set then it will or the address with 0xFFFF0000 to get the I/O processor's address.

32.0.7 *SPOOL

Description Writes all content sent to the screen to a file on disc.

This command is provided by the Operating System so is available to all file systems.

33 Cassette Filesystem

Page intentionally blank

34 ROM Filesystem

Page intentionally blank

35 Acorn DFS

The original Acorn Disk File System (DFS)[16] is a single sided, single density 40 or 80 track format. A double sided disc can hold two DFS volumes. Each track carries ten 256 byte sectors numbered from 0 to 9.

[16] *Acorn DFS disk format*, beebwiki.mdfs.net (Feb. 25, 2023)

35.1 Commands

35.1.1 *ACCESS

Description Lock/unlock a file to prevent it being deleted or overwritten

Syntax *ACCESS file L

*ACCESS file

Once a file is locked you cannot delete, overwrite or write to a locked file until you unlock it again. If you load a file which is locked you will not be able to save it again with the same name.

Note: Locking a file does not prevent the file being removed from a disc when it's formatted with either *FORM40[1], *FORM80[2] or overwritten with *BACKUP[3].

1: *FORM40 **??**
2: *FORM80 **??**
3: *BACKUP
Subsection 35.1.2 on page 792

35.1.2 *BACKUP

This command can overwrite memory so be certain to save your work first before executing this command.

Description Backup a disk to another disk producing an identical copy.

Syntax ***BACKUP** src dest

Example *ENABLE
BACKUP 0 1

This command requires *ENABLE[4] to be called first. If this is not done then the error `Not Enabled` will be issued.

4: *ENABLE
Subsection 35.1.12 on page 796

35.1.3 *BUILD

Description Create a file directly from the keyboard

Syntax ***BUILD** !boot

Example *BUILD !BOOT

This command can be used to create exec files and the !BOOT file used with SHIFT+BREAK.

When this command is executed, line numbers are displayed. Typing ESCape on a line by itself will terminate the command.

```
>*BUILD !BOOT
00010 first line of text
00020 second line
00030 ESC
```

35.1.4 *CAT

Description Display the disc catalogue

Syntax *CAT 0

Short *.

Example The last two lines are identical
```
CAT 0
CAT
.
```

This command is used to show the files present on a disc. If the drive number is omitted then the current drive is assumed.

35.1.5 *COMPACT

Description Move all files to the start of the disc creating a single continuous block of free space at the end

Syntax *COMPACT drive

This command can overwrite memory so be certain to save your work first before executing this command.

As files are saved they use up free space on the disc. When they are deleted this frees up that space but, if other files were saved after the first one, then the free space is broken into multiple blocks.

As DFS writes a file in a single block, if there isn't a block of free space large enough then a `Disc full` error will be issued, even if there is logically enough space available on the disc.

This is where *COMPACT comes in to play. It will take every file on the disk and move them to the beginning, ensuring that all the free space on the disk is at the end and in a single block.

If there are no spaces between files on the disc, then this command does nothing.

35.1.6 *COPY

Description Copy a named file or files from one disc to another

Syntax *COPY src-drv dst-drv filename

Example *COPY 0 1 HELLO

This command can overwrite memory so be certain to save your work first before executing this command.

The example above will copy the file HELLO in the current directory on drive 0 onto drive 1.

Wildcards are also supported. For example,

```
*COPY 0 1 #.MY*
```

will copy all files beginning with **MY** regardless of which directory they are in.

Anything already existing on the destination drive will not be affected.

35.1.7 *DELETE

Description Delete a single file from the disc.

Syntax *DELETE filename

Example *DELETE FRED

Once a file has been deleted, it cannot be restored.

If the file is locked then the error `File locked` is issued.

If the disc is write protected then the error `Disc write protected` is issued.

If the file does not exist then the error `File not found` is issued.

See *DESTROY[5] to delete multiple files.

5: *DESTROY
Subsection 35.1.8 on page 795

35.1.8 *DESTROY

Description Delete multiple files from a disc in a single action.

Syntax ***DESTROY** filename

Example
```
*ENABLE
DESTROY *.H*
A.HELLO
B.HELLO
Delete (Y/N)?
```

Once a file has been deleted, it cannot be restored.

***ENABLE** must be issued immediately before this command. If this is not done then the error `Not enabled` is issued.

If the disc is write protected then the error `Disc write protected` is issued.

If a file is locked then the file is ignored.

See *DELETE[6] to delete single files.

6: *DELETE Subsection 35.1.7 on page 794

35.1.9 *DIR

Description Set the current directory

Syntax ***DIR** A

This command will set the current directory[7].

7: In DFS, directories are single letters, not names.

35.1.10 *DRIVE

Description Set the current drive

Syntax ***DRIVE** 1

This command will set the current drive. Once set, commands that can optionally work without specifying a drive will use this drive.

35.1.11 *DUMP

Description Produce a hexadecimal listing of a file on the screen

Syntax ***DUMP** filename

See also *TYPE[8], *LIST[9]

This command will write a file to the screen in hexadecimal.

8: *TYPE ??
9: *LIST Subsection 35.1.16 on page 797

35.1.12 *ENABLE

Description Enable DFS commands which are dangerous

Some commands can have irreversible effects so are protected from normal use by requiring this command to be executed first. These are `*BACKUP`[10] and `*DESTROY`[11]. If this command is not executed before either of those commands then the error `Not enabled` is issued.

10: `*BACKUP` Subsection 35.1.2 on page 792
11: `*DESTROY` Subsection 35.1.8 on page 795

35.1.13 *EXEC

This command is provided by the operating system and is defined in full within section Subsection 32.0.2 on page 784.

35.1.14 *INFO

Description Display information about a file or group of files

Syntax ***INFO** filename

Example *INFO A.HELLO
A. HELLO L 001900 00801F 00003B 003

If the file is not found then the error `File not found` is issued.

Wildcards # and * can be used to display information for groups of files.

35.1.15 *LIB

Description Set the library to the specified drive and directory

Example *LIB :1.A

This command sets the library path to the specified drive and directory. Then, when a ***filename*** command is issued, that directory will be searched for a file of that name and will be loaded and executed.

e.g. with the above example, typing **`*MYFILE`** would be the same as if you had issued **`*RUN :1.A.MYFILE`**

The purpose of this facility is to have commands on one disk which can then be used to manipulate files on another "data" disk.

35.1.16 *LIST

Description Display a text file on screen with line numbers

Example *LIST DATA

See also *TYPE[12], *DUMP[13]

This command will write the contents of a file to the screen including line numbers.

12: *TYPE **??**
13: *DUMP
Subsection 35.1.11 on page 796

35.1.17 *LOAD

This command is provided by the operating system and is defined in full within section Subsection 32.0.3 on page 784.

35.1.18 *OPT 1

This command is provided by the operating system and is defined in section Subsection 32.0.4 on page 781.

However, the parameters for this command are filesystem specific.

Description Display a message every time a file is accessed

Syntax *OPT 1,n

Example *OPT 1,1

This command enables or disables a message system which displays a files information similar to `*INFO`[14] every time a file on the disc is accessed.

This is disabled if n=0, but enabled when n is within 1...99

14: `*INFO` Subsection 35.1.14 on page 796

35.1.19 *OPT 4

This command is provided by the operating system and is defined in section Subsection 32.0.4 on page 781.

However, the parameters for this command are filesystem specific.

Description Change the auto-start option for the current disc

Syntax *OPT 4,n

Example *OPT 4,1

This command sets the auto-start option of the disk which takes effect if the disk is in drive 0 when SHIFT BREAK is pressed. Table Table 35.1 shows the valid values of *n*.

Table 35.1: Acorn DFS *OPT4 parameters

n	Action on SHIFT+BREAK
0	Does nothing
1	***LOAD !BOOT**
2	***RUN !BOOT**
3	***EXEC !BOOT**

35.1.20 *RENAME

Description Change a filename and move directory if required

Syntax *RENAME old new

Example *RENAME SUMS B.MATHS

Renames a file optionally changing its directory at the same time. It will fail if a drive number is provided and the drive numbers differ between the old and new file names.

In the above example, if the current directory is `$` then the file `$.SUMS` is renamed to `B.MATHS`.

35.1.21 *RUN

This command is provided by the operating system and is defined in full within section Subsection 32.0.5 on page 786.

35.1.22 *SAVE

This command is provided by the operating system and is defined in full within section Subsection 32.0.6 on page 786.

35.1.23 *SPOOL

This command is provided by the operating system and is defined in full within section Subsection 32.0.7 on page 786.

35.2 DFS Internals

35.2.1 Sector ID's

Each sector is given an ID at format time; the parameters of the ID, listed below, are used by the DFS and the floppy disc controller to address sectors.

- C the cylinder number ranges from 0 on the outermost track to 39 or 79 on the innermost track.
- H the head number = 0. Acorn DFS ignores this after formatting. Some DFS formatters set it to 1 on the upper surface of the disc (corresponding to drives 2 or 3.)
- R the record number runs sequentially from 0 to 9 within each track. There is a standard track skew of 3; that is, the sectors having R = 0 are the first sector of the first track, the fourth sector of the second track, the seventh of the third, the tenth of the fourth, and so on.
- N the record length = 1, showing that the sector length is 27 + 1, or 256 bytes.

All these parameters can potentially take any value between 0 and 255, and so confuse Acorn DFS and some non-standard controllers. Some copy protection schemes make use of this feature.

35.2.2 Catalog

The catalog is located in sectors 0 and 1 on Track 0. It is capable of holding just 31 file entries. Figure Figure 35.1 on page 801 shows the content of the DFS Catalog[17].

Formatting a disk on the real hardware only needs to write to these two sectors (via '*FORM40' or '*FORM80') but a blank catalogue simply needs these two sectors blanked and the disk size field initialised.

[17] *Acorn, Watford, Solidisk and Duggan DFS Filesystem Structure*, mdfs.net (Feb. 25, 2023)

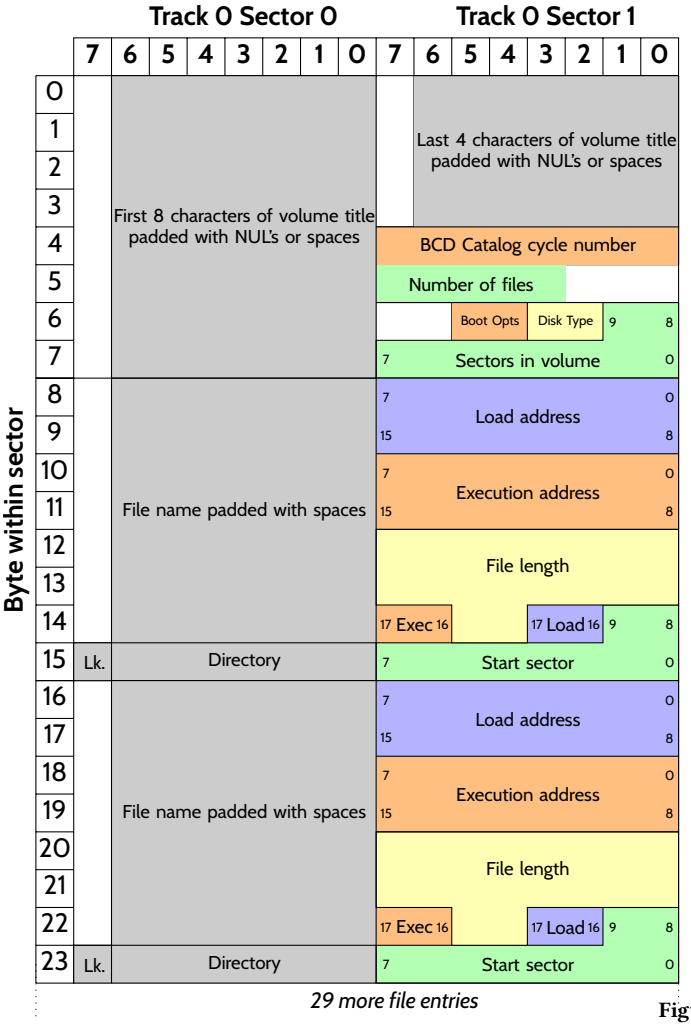


Figure 35.1: Acorn DFS Catalog Format

Catalog header

Disc title Up to 12 printable ASCII characters, padded with NULs (0x00) or spaces. The first eight characters are placed in sector 0 bytes 0 to 7, the last four characters in sector 1 bytes 0 to 3.

Cycle number A binary-coded decimal (BCD) value where each hex nibble takes a value from 0 to 9. This starts at 0x00 and is incremented each time the catalogue is rewritten.

It provides a simple version control system and allows applications to confirm that their internal copies of the catalogue are up-to-date.

Number of files The number of files counter is stored in sector 1 byte 5 as bits 3…7.

Bits 0…2 are 0 making the entire byte the offset to the last valid entry in the catalog.

There may be up to 31 files on the disc.

Boot option Selects the action to be taken when the disc is booted. This can be set using `*OPT 4`[15]. The field uses two bits of byte 6 in sector 1. Table Table 35.2 shows the possible options.

15: `*OPT 4` Subsection 35.1.19 on page 798

Table 35.2: Boot action

Bit 5	Bit 4	Action
0	0	None
0	1	*LOAD $.!BOOT
1	0	*RUN $.!BOOT
1	1	*EXEC $.!BOOT

Disk Type Unused in standard DFS, these two bits in byte 6 of sector 1 can be used to detect third party versions of DFS. Table Table 35.3 shows the types of Disk

Table 35.3: Disk Type

Bit 3	Bit 2	Filesystem
0	0	Acorn DFS and Watford DFS up to 256K
0	1	Watford DFS over 256K
1	0	Duggan HDFS, single sided
1	1	Duggan HDFS, double sided

Disc size The total number of sectors on this side of the disc. This is a 10-bit field with the low 8 bits in sector 1 byte 7, and the others in the low two bits of sector 1 byte 6.

File descriptor fields

There is room for 31 file entries in the catalogue. Each entry takes up eight bytes in each of the two catalogue sectors.

Files must be listed in descending order of start sector with no gaps in the catalogue. Empty files should be given a start sector of 2. Files cannot be fragmented but there may be free sectors between files.

The rest of each sector represents the files. Each file is represented by a block of 8 bytes in each sector offset by 8. So the last file starts at offset 8, second to last at 16 and so on.

File name One to seven valid file name characters stored in sector 0 bytes 8 to 14.
Valid characters are the printable ASCII characters between 0x20 and 0x7E inclusive, except . : " # * and space.
The field is padded with spaces.
Bit 7 should always be clear but are used for other file systems as extra storage, either adding a bit to another field such as WDFS or file permissions as in HDFS.
When opening a file the DFS searches for it by name.

Directory One valid file name character stored in the low 7 bits of sector 0 byte 15 Identifies the directory (namespace) to which the file belongs.
DFS directories are like those in ADFS, DOS or Linux except they cannot nest (they are all root directories) and they share the volume catalogue.
The combination of file name and directory must be unique in the volume.

Locked Attribute If the top bit of the directory is set, then the file is locked and may not be altered or deleted. This bit is set using the `*ACCESS`[16] command.

Load address An 18 bit address in memory where the file should be *LOADed by default. The low 8 bits are in sector 1 byte 8; the next 8 bits in sector 1 byte 9; the top two bits are in bits 2 and 3 of sector 1 byte 14.

If the file is not meant for *LOADing (*e.g.* a sequential file or text) all 18 bits should be set.

Execution address An 18 bit address in memory to be jumped to when the file is *RUN. This need not be within the bounds of the loaded file. The low 8 bits are in sector 1 byte 10; the next 8 bits in sector 1 byte 11; the top two bits are in bits 6 and 7 of sector 1 byte 14.

If the file is a sequential file or text then all 18 bits should be set.

16: `*ACCESS`
Subsection 35.1.1 on page 791

Note on load and execution addresses

If bits 16 and 17 of the address are both set, the address refers to I/O processor memory and OSFILE will return the address ORed with 0xFFFF0000 in its parameter block.

Otherwise it is an address in second processor memory and OSFILE will return the address as is.

In practice the bits are sometimes found clear when they should be set.

File length An 18 bit value giving the number of bytes in the file.
The low 8 bits are in sector 1 byte 12; the next 8 bits in sector 1 byte 13; the top two bits are in bits 4 and 5 of sector 1 byte 14.

Start sector The 10 bit logical block address of the first sector that contains the file. The low 8 bits are in sector 1 byte 15; the top two bits are in bits 0 and 1 of sector 1 byte 14. The cylinder and record numbers are the result and remainder, respectively, when the LBA is divided by 10.
A start sector of 0 or 1 is invalid as the file would overlap the catalogue.

Issues with early versions of DFS

Some early DFSs allow the user to write disk titles or filenames with bit 7 set.

This may cause these entries to appear to erroneously have HDFS or WDFS attributes.

Most DFSs disallow top-bit set filenames and disk titles.

35.2.3 Identifying and validating a DFS disc image

Unlike its successor ADFS, the DFS format contains no self-identification strings or checksums, so recognising a disc or disc image as Acorn DFS is a complex and unreliable process.

If one has a floppy disc in hand it is often enough to confirm that the physical format is the same (see above) as Acorn DFS is the majority producer. On the other hand DFS disc image files are normally given the extension .ssd or .dsd, indicating a single or double sided disc image respectively.

Often the task is to identify which format variant a disc or image contains. This is more a matter of finding valid catalogues in the appropriate locations.

To check that a pair of sectors contains a valid standard DFS catalogue

- Firstly confirm that all bits marked white or cream in the above diagram are zeroes; if not, the catalogue can be rejected immediately.
- Check also that the file offset is a multiple of 8.
- The disc size must be between 2 and 800 sectors inclusive, although the format allows up to 1023 sectors, and disk images may contain up to this many.
- The disc title consists of printable ASCII characters, padded with NULs or spaces.
- For each file entry in use:
    - The file name field must contain one to seven valid characters and be padded with spaces. Valid characters are the printable ASCII characters between 0x20 and 0x7E inclusive, except . : " # * and space.
    - The directory character (without the attribute bit) must be a valid file name character.

- The file name (taking the directory character into account) must be unique.
- The start sector must be more than 1 and less than the disc size.

▸ Skipping over zero-length files:

- The start sectors must run in strictly descending order.
- No file may overlap the one before it in the catalogue. That is:
  $startSector_2 + \frac{length_2+255}{256} \leq startSector_1$
- No file may overshoot the end of the disc. That is:
  $startSector + \frac{length+255}{256} \leq discSize$
  Equivalently, the first file in the catalogue must not overlap the imaginary file whose start sector field is the disc size field.

Further heuristics Any catalogue complying with the above is a valid DFS catalogue; however, only a tiny subset of these look like catalogues seen in real life. As an informal test a pair of sectors that match many of the following is more likely to be a catalogue:

- The disc size is 400 sectors (40 tracks) or 800 sectors (80 tracks.)
- There is a file named !BOOT in directory $.
- The 16th and 17th bits of the load and execution (exec) addresses are all clear or all set in each file entry.
- There are few directories, or a few unique names in several directories.
- Most directory characters are $ (the default.)
- Some files have load addresses of the form 3nn00 (nn=0E, 11, 19, 1B, 1D, 1F or 21) and exec addresses of 0x3801F or 0x38023[17]
- Some files have load and exec addresses the same, usually xx00 or 3xx00, and often have length yy00[18]
- Some files have load and exec addresses of 0, `0x30000` or `0x3FFFF` [19]
- Many files are less than 2K; few files more than 20K; no files more than 64K except text and sequential data.
- The file names have very few lowercase letters.
- The file names contain commonly-occurring pairs of letters.

17: BASIC programs

18: Screen dumps and machine code images

19: Text and sequential data

Page intentionally blank

36 Acorn ADFS

Page intentionally blank

LanManager File System 37

The Sprow ethernet[18] card is an ethernet interface which is installed inside the Master 128 where the Econet interface would normally be installed. A custom Econet to RJ45 Ethernet cable[1] then connects the Master 128 to an Ethernet switch.

The `*LANMAN`[2] command selects this file system.

[18] *Sprow Master 10/100 Net User Guide*, (2008)

1: Econet to RJ45 Ethernet cable Appendix F.5.1 on page 1002

2: `*LANMAN` Subsection 37.3.12 on page 822

37.1 About the SPROW 10/100 Network interface

The SPROW 10/100 Network interface is a small board for the BBC Master series which sits in the location normally used for the Econet interface. Instead of providing Econet, this board supplies a 10/100Mb Ethernet interface and supports both TCP/IP V4 and CIFS/LanManager/SMB file sharing.

It comes with a ROM which provides the underlying TCP/IP network layer as well as the LANMANAGER file system which operates over the network layer.

The interface board itself also holds 76K of RAM for packet buffers, 1K of non-volatile memory for holding configuration data.

37.2 Networking Commands

This section lists the commands provided by the LANMANAGER ROM which deal with raw TCP/IP Networking.

37.2.1 *CONFIGURE EMLINK

Description Configure the hardware link speed

Syntax *CONFIGURE EMLINK AUTO | 10 | 100 Half | Full

short *CONFIG.EMLINK

Sets the hardware link speed. Ordinarily this should be set to AUTO so the network module will negotiate the best link speed with whatever it is plugged into.

Manually configuring the interface may mean that no link will be established if the link partner cannot support the chosen option(s). It also means that the network module can no longer detect and correct for cross wired cables.

37.2.2 *CONFIGURE EMADVERTISE

Description Configure the capabilities declared to connected devices

Syntax ***CONFIGURE EMADVERTISE** [*10 [HALF FULL]*] [*100 [HALF FULL]*]

short ***CONFIG.EMADVERTISE**

Sets the capabilities that are declared to anything that the network cable is plugged into. This can selectively declare a 10 or 100 Mbps link with either half for full duplex. If neither HALF or FULL is specified then both will be advertised.

By only advertising a reduced set of capabilities the link rate can be reduced while allowing the EMLINK[3] configuration option to be left as AUTO.

3: EMLINK
Subsection 37.2.1 on page 814

37.2.3 *CONFIGURE EMADDR

Description Configure the IP address

Syntax ***CONFIGURE EMADDR** [*AUTO*] | n.n.n.n

37.2.4 *CONFIGURE EMMASK

Description Configure the IP network mask

Syntax ***CONFIGURE EMMASK** [*AUTO*] | n.n.n.n

37.2.5 *CONFIGURE GATEWAY

Description Configure the network gateway address

Syntax ***CONFIGURE GATEWAY** [*AUTO*] | n.n.n.n

37.2.6 *CONFIGURE DNS

Description Configure the DNS server address

Syntax ***CONFIGURE DNS** n.n.n.n [*n.n.n.n*]

37.2.7 *EMINFO

Description List information about network module's state

Syntax *EMINFO -verbose

short *EMI.

See also *HELP LANMANAGER[4]

This command lists information about the network module's running state including more information than `*HELP LANMANAGER` provides.

The option -verbose includes some counters useful for diagnosing networking issues.

4: *HELP LANMANAGER
Subsection 37.3.11 on page 822

37.2.8 *EMTEST

Description Perform hardware self tests

Syntax *EMTEST

short *EMT.

37.2.9 *PING

Description Ping a remote IP address

Syntax *PING -quiet address | name

short *PI.

Sends an ICMP message to the given address or host name. The ping will stop when ESCAPE is pressed.

The -quiet option will only show a summary once ESCAPE has been pressed.

37.2.10 *STATUS EMLINK

Description Show current hardware link settings

Syntax *STATUS EMLINK

short *ST.EMLINK

37.2.11 *STATUS EMADVERTISE

Description Show current advertised capabilities

Syntax *STATUS EMADVERTISE

short *ST.EMADVERTISE

37.2.12 *STATUS EMADDR

Description Show current IP Address

Syntax *STATUS EMADDR

short *ST.EMADDR

37.2.13 *STATUS EMMASK

Description Show current network mask

Syntax *STATUS EMMASK

short *ST.EMMASK

37.2.14 *STATUS GATEWAY

Description Show current gateway address

Syntax *STATUS GATEWAY

short *ST.GATEWAY

37.2.15 *STATUS DNS

Description Show current DNS settings

Syntax *STATUS DNS

short *ST.DNS

37.3 CIFS Commands

This section lists the commands provided by the LANMANAGER ROM which deal with the actual file system.

37.3.1 *ACCESS

Description Set a files attributes

Syntax *ACCESS L | R | W

short *AC.

Sets a files attributes to be Locked, Readable or Writeable, although all files will always be readable with CIFS.

37.3.2 *BYE

Description Alias for *DISMOUNT

37.3.3 CDIR

Description Create a directory

Syntax *CDIR directory

short *CD.

37.3.4 *COPY

Description Copies one or more files to the given drive/dir pair.

Syntax *COPY afsp :drive.directory

Copies one or more files to the given drive/dir pair.

To make a duplicate of a file within the same directory use the MOS `*MOVE` command.

37.3.5 *DESTROY

Description Delete multiple files

Syntax *DESTROY afsp

short *DES.

See also *WIPE[5]

Delete files, prompting for confirmation for the entire batch.

5: *WIPE
Subsection 37.3.17 on page 824

37.3.6 *DISMOUNT

Description Dismount a remote shared folder, closing any open files

Syntax *DISMOUNT

37.3.7 *DRIVE

Description Provided for compatibility with DFS[6]

Syntax *DRIVE drive#

short *DR.

This command is included only for compatibility with files transferred from the disc filing system.

6: DFS Chapter 35 on page 791

37.3.8 *DIR

Description Set the current directory

Syntax *DIR drive:.directory

37.3.9 ENABLE

Description Enable dangerous commands

Syntax *ENABLE

short *ENA.

37.3.10 *FREE

Description Show free space statistics

Syntax *FREE

short *FR.

37.3.11 *HELP LANMANAGER

Description Report version and status information

Syntax *HELP LANMANAGER

short *H.LANMANAGER

See also *EMINFO[7]

Reports the version of the networking software along with a summary of the network status.

7: *EMINFO
Subsection 37.2.7 on page 817

37.3.12 *LANMAN

Description Select the LanManager file system

Syntax *LANMAN

37.3.13 *LIB

Description Set current library

Syntax *LIB drive:.directory

When a share is first mounted the library will be set to ”Library1” is it exists, otherwise the library will default to the root, ”$”.

37.3.14 *MOUNT

Description Mount a remote shared folder

Syntax *MOUNT share [*username password [workgroup]*]

short *MOU.

Example *MOUNT \\OfficePC\BeebGames

*MOUNT \\192.168.1.54\BeebGames

Attempts to connect to the given share, which will then be referred to as drive 0. If the username or password is omitted then it will attempt to connect with a guest[8] account.

Where the share to connect to is not in the default workgroup then the workgroup[9] must also be specified.

8: Guest accounts usually get reduced functionality.

9: The default workgroup is `WORKGROUP`

37.3.15 *RENAME

Description Rename the given file or directory

Syntax *RENAME oldfsp newfsp

short *REN.

37.3.16 *SDISC

Description Alias for *MOUNT

37.3.17 *WIPE

Description Delete multiple files

Syntax *WIPE afsp

short *W.

See also *DESTROY[10]

Delete files, prompting for confirmation for each one.

10: *DESTROY
Subsection 37.3.5 on page 821

38 Network File System

Page intentionally blank

39 Alternative File Systems

39.1 Acorn System DFS

Acorn System DFS is also known as Atom DOS

Discs for the System series have an identical catalogue format to those for the BBC Micro in section Chapter 35 on page 791, except all entries are in a directory " " (space), disks do not have a cycle number, and files do not have bits 16 and 17 of load and execution addresses[16].

[16] *Acorn DFS disk format*, beebwiki.mdfs.net (Feb. 25, 2023)

The places they would occupy are reserved and must be cleared. A BASIC program named DCONV, on the Utility disc, performs this task.

39.1.1 Differences between Acorn DFS and System DFS

- The Reserved entries must be clear.
- The disk type in bits 2 and 3 of sector 1 byte 6 must be clear as with DFS.
- Lk. in byte 7 is the same as for Acorn DFS

39.2 Angus Duggan HDFS

Angus Duggan's HDFS shares the same basic file structure as Acorn's DFS[16] in Chapter 35 on page 791.

[16] *Acorn DFS disk format*, beebwiki.mdfs.net (Feb. 25, 2023)

Sector 1 byte 6 bit 3 is set for HDFS with bit 2 clear for single-sided and set for double-sided disks.

HDFS subdirectories follow the same format as the main catalogue.

The places they would occupy are reserved and must be cleared. A BASIC program named DCONV, on the Utility disc, performs this task.

39.2.1 Catalog

The catalog is located in sectors 0 and 1 on Track 0. It is capable of holding just 31 file entries. Figure 39.1 on page 829 shows the content of the DFS Catalog[17].

Formatting a disk on the real hardware only needs to write to these two sectors[1] but a blank catalogue simply needs these two sectors blanked and the disk size field initialized.

[17] *Acorn, Watford, Solidisk and Duggan DFS Filesystem Structure*, mdfs.net (Feb. 25, 2023)

1: via `*FORM40` or `*FORM80`

39.2.2 Differences between Acorn DFS and HDFS

- For the volume header, bytes 0 … 7 of both sectors:
  - Track 0 sector 0 byte 0 bit 7 is bit 10 of the number of sectors in the volume.
  - Track 0 sector 1 byte 6 bit 3 is set.
  - Track 0 sector 1 byte 6 bit 2 is the number of sides-1: 0=single-sided, 1=double-sided.
- For file entries:
  - *Sect* is bit 10 of the start sector.
  - *Len* is bit 18 of the file length.
  - *IsDir* defines if the entry is a file or a subdirectory.
  - If the $\overline{Read}$, $\overline{Write}$ or $\overline{Exec}$ attributes are set then the file can not be read, written or executed respectively.

  For a directory entry, the directory has the same layout as the main Catalog, except the volume title becomes the directory name.

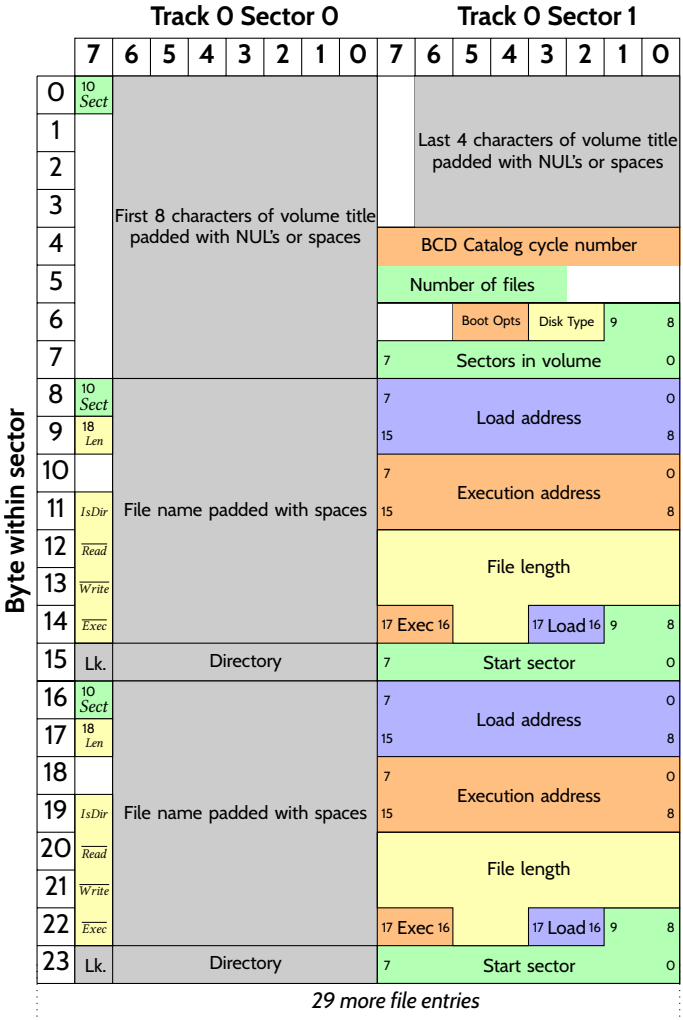


Figure 39.1: Duggan HDFS Catalog Format

39.3 Opus DDOS

This is a double density format, but it is mentioned here as the volume catalogues are based on Acorn DFS[16] in section Chapter 35 on page 791.

[16] *Acorn DFS disk format*, beebwiki.mdfs.net (Feb. 25, 2023)

DDOS discs are partitioned into between one and eight volumes. Track 0 is reserved for the DFS-style volume catalogues, plus a disc catalogue unique to DDOS.

In each volume catalogue:

- The start sector may be any number less than the disc size, as DDOS adds a partition offset and so the catalogue is not in the same space as the files.
- The disc size is the size of the current volume. It is an 11 bit quantity, the top bit being in bit 2 of sector 1 byte 6.
- If and only if this bit is set:
- The file length steals an extra bit from the execution address of each file (bit 6 of sector 1 byte 14+n.)
- The start sector steals an extra bit from the load address of each file (bit 2 of sector 1 byte 14+n.)

39.4 Watford DFS

The Watford DFS uses the same structure as Acorn's DFS with some additional attributes:

In byte 6 of Sector 1, bit 2 is clear and 3 is clear for Acorn and WDFS for disks up to 256K, but set for WDFS and disks larger than 256K.

Watford DFS use 19-bit file lengths and 11-bit sector numbers making the largest possible disk being 512K and the largest file size of 512K-1.

39.4.1 Catalog format

The catalog is almost identical to Acorn DFS with a couple of minor changes as shown in figure Figure 39.2 on page 832 shows the content of the DFS Catalog[17].

[17] *Acorn, Watford, Solidisk and Duggan DFS Filesystem Structure*, mdfs.net (Feb. 25, 2023)

Formatting a disk on the real hardware only needs to write to these two sectors (via '*FORM40' or '*FORM80') but a blank catalogue simply needs these two sectors blanked and the disk size field initialized.

39.4.2 Differences between Acorn DFS and WDFS

- Sector 1 byte 6 bit 3 is always 0 like Acorn DFS
- Sector 1 byte 6 bit 2 is bit 10 of the disk size / sectors in volume. 0=max 256K, 1=max 512K
- Len in byte 5 of the filename is bit 18 of the File Length.
- Len in byte 6 of the filename is bit 10 of the Start Sector.
- Lk. in byte 7 is the same as for Acorn DFS

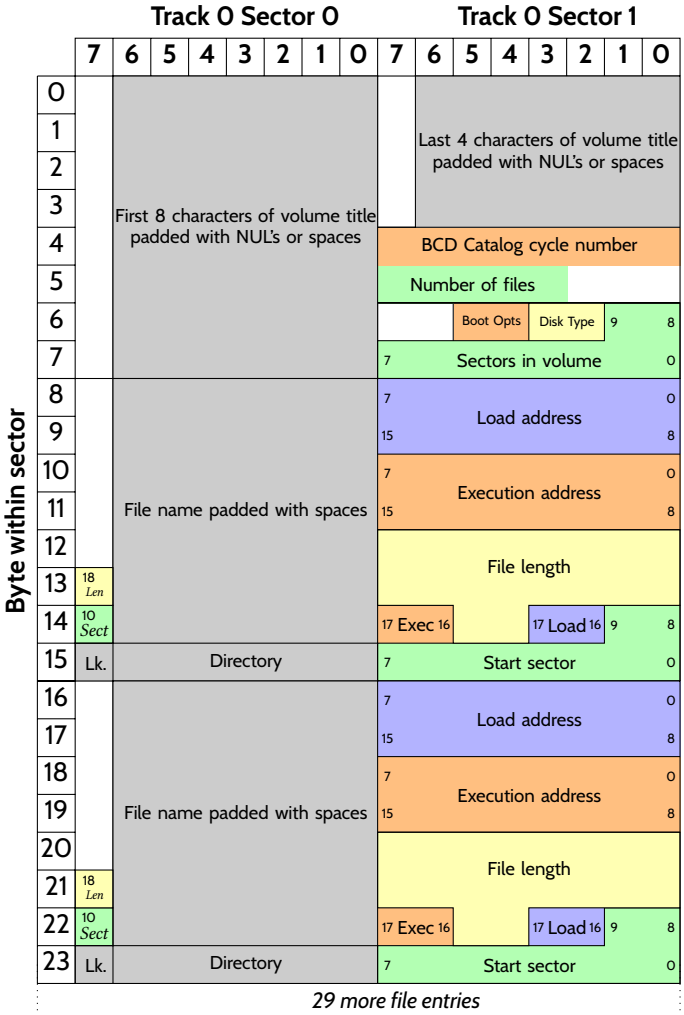


Figure 39.2: Watford DFS Catalog Format

39.5 Watford DDFS

This is also a double density format with catalogues similar to Acorn DFS.

It uses 19 bit file lengths and 11 bit LBAs throughout, but the 'extra' bits are stored in the top bits of the disc title and file names, which are cleared in Acorn DFS.

39.6 Watford 62 Disk File System

Watford 62 DFS is identical to Watford DFS except it places a second catalogue in sectors 2 and 3 of track 0, allowing 62 files on one volume. To identify the second catalogue it sets its title to 12 × 0xAA bytes.

Without due care a system running Acorn DFS can corrupt a Watford disc as it is unaware of the extra files. On such a system the user may run a ***SWAP** utility to exchange the catalogues. In practice there is a special entry in the first catalogue (directory !, name !!!!!!!, length 0x200, start sector 2) protecting the second one from being overwritten.

If any file in the first catalogue starts before sector 4, and it is not the special file, the disc is not a Watford DFS disc.

39.7 Telesoft File System

39.8 IEEE File System

The Commodore C64

Table of contents

40 Kernal **837**
40.1 Spelling of Kernal 837

41 Memory Layout **839**
41.1 Colour RAM 839

42 Serial Bus **841**
42.1 Types of Device 841
42.2 Addressing 842
42.3 Serial Bus Cable 842

40 Kernal

The Kernal is the operating system for the Commodore 64.

40.1 Spelling of Kernal

Lets get this out of the way. The operating system is spelt **Kernal** and not Kernel.

From what I've read, the spelling mistake occurred in the late 1970's when the OS was being written for the Commodore PET and was a typing mistake which was never corrected.

Page intentionally blank

41 Memory Layout

41.1 Colour RAM

For each character in every row (40 chars per row; 0-39) and column (25 columns; 0-24) one of 16 colors can be assigned. The color values 0-15 are only used in the lower nibbles. The upper nibbles are undefined and are purely random values. Figure Figure 41.1 on page 839 shows the layout of this memory The colour memory is provided by an MM2114N-3 4Kb (1024x4) static RAM chip.

As this chip only provides 4 bits, the upper nibble is random as it's not connected to anything. It is separate from the onboard 64K RAM



Figure 41.1: C64 Colour RAM layout

Page intentionally blank

42 Serial Bus

The serial bus[19] is a bus where devices can be daisy-chained together to allow for peripherals to be connected over a single cable. The advantage of the serial bus is that more than one device can be connected to the port. Up to 5 devices can be connected to the serial bus at one time.

[19] *Commodore 64 Programmers Reference Manual*, () Ch.6 P.362

If this sounds familiar, think USB.

USB, or "Universal Serial Bus" is a serial bus where devices are connected over a single cable.
A USB Hub is a device on the bus so devices connected via one is daisy-chained with it.

42.1 Types of Device

There are three types of operation over the serial bus, Control, Talk and Listen.

- A Controller device is one which controls the serial bus.
- A Talker is a device which transmits data onto the bus.
- A Listener is a device which listens for data sent over the bus.
- Only one device can be a Talker, otherwise the devices would collide with each other when they transmit their data.
- Multiple devices can be Listeners at the same time.
- All devices connected to the serial bus receives all the data transmitted over it.

The computer is the controller of the bus. It acts as a Talker when it sends data, *e.g.* to the Printer or the Floppy Drive, and as a Listener when receiving data from a device like the Floppy Drive.

Printers are only Listeners, but the Floppy Drive is both a Talker and a Listener.

42.2 Addressing

To enable the controller to target the destination device, each device has a bus address. Addresses on the serial bus range from 4 … 31.

Each address has its own unique address. However, some devices also have a secondary address used for configuration.

For example, to open a connection on the bus to the printer, but have it print in Upper and Lower case, then:

```
OPEN 1,4,7
```

where:

1 is the logical file number, *e.g.* the number you `PRINT#` to.
4 is the address of the printer
7 is the secondary address telling the printer to enter Upper/Lower case mode.

42.3 Serial Bus Cable

Each device on the bus has two Serial Bus ports, one each to connect to an upstream and downstream device. Only the controller has a single port as it's at the start of the bus.

The serial bus itself is a 5-wire cable which uses a 6-pin connector. Appendix G.1 on page 1013 shows the layout of the serial bus ports.

The Sinclair ZX Spectrum

Table of contents

43 Spectrum Keyboard **845**

44 Spectrum Memory Map **847**
44.1 Memory Map Overview 847
44.2 Memory Contention 848
44.3 128K and +2 Memory Map 849
44.4 Spectrum +2A and +3 Memory Map 851

45 File Formats **855**
45.1 TAP Format 855

46 Machine Code **857**

43 Spectrum Keyboard

The keyboard on the various models of the Spectrum share the same underlying keyboard matrix meaning that the keyboard is treated virtually identicaly between models.

Appendix I.1 on page 1021 lists all of the keys available on the Spectrum along with both the port address and which bit on that address the key exists.



Figure 43.1: Spectrum 48K Keyboard

Page intentionally blank

44 Spectrum Memory Map

44.1 Memory Map Overview

Figure Figure 44.1 on page 847 shows an overview of the Spectrum Memory map for both 48K and 128K/+2 machines.

The +2A and +3 are similar to the 128K but have additional ROM's and alternate all RAM memory layouts.

The specifics for the individual machines are listed in their own sections.

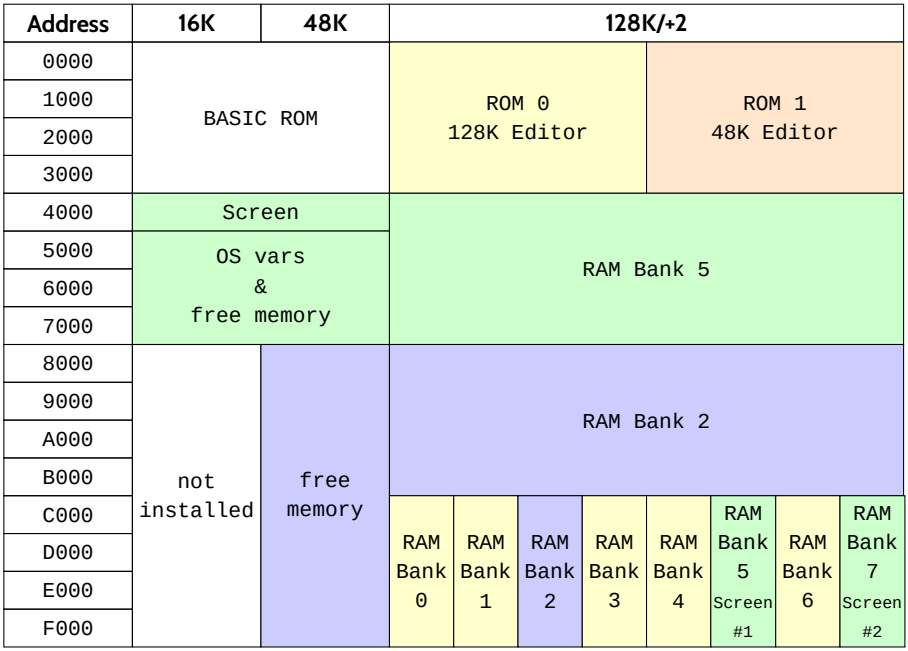


Figure 44.1: Spectrum Memory Map Overview

44.2 Memory Contention

Some memory is contended with the ULA, with the latter having priority whenever the screen is being updated. Time critical code needs to run in uncontended RAM.

On the 48K memory between 0x4000…0x7FFF is contended with the ULA whilst 0x8000…0xFFFF is not.

On the 128K and +2, odd numbered RAM banks are contended with the ULA. Time critical code should run in even numbered RAM banks.

On the 16K, all memory is contended as there is no memory installed in the range 0x8000…0xFFFF.

44.3 128K and +2 Memory Map

This section pertains to the 128K and +2 machines.

The +2A and Spectrum +3 machines use a different memory layout which is described in Section Section 44.4 on page 847.

The memory layout of the Spectrum 128K and Spectrum +2 machines is shown in Figure Figure 44.1.

It consists of:

- two 16K ROM's which can be paged in to address 0x0000
- Bank 5 available between 0x4000...0x7FFF
- Bank 2 available between 0x8000...0xBFFF
- Banks 0...7 pageable between 0xC000...0xFFFF

In normal use, 128 BASIC configures the memory as:

- Bank 0 paged in to `0xC000`
- Banks 1, 3, 4, 6 and most of 7 used for the RAM Disk.
- Some of Bank 7 is used for the editor scratch pad

44.3.1 Switching RAM and ROM's

The RAM banks and ROMs are paged by writing to port `0x7FFD`. This port is write only so a copy is also stored in RAM at address `0x5B5C`.

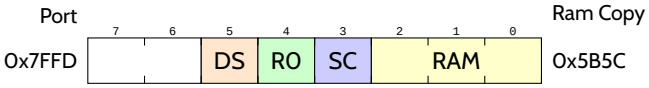
Figure 44.2 on the next page shows the structure of this port, whilst Listing 44.1 shows an example of how to switch RAM banks.

Table 44.1: Spectrum 128K/+2 Paging Port

Port	RAM Copy
`0x7FFD`	`0x5B5C`

```
1    LD  A,(0x5B5C)      ; Get previous port
     value
2    AND 0xF8
3    OR  4               ; Select bank 4
4    LD  BC,0x7FFD       ; Port address
5    DI                  ; Disable interrupts
6    LD  (0x5B5C),A      ; Update memory copy of
     port
7    OUT (C),A
8    EI                  ; Enable interrupts
```

Listing 44.1: Switching memory bank on a 128K Spectrum



RAM The RAM bank to page into 0xC000…0xFFFF

SC The screen to display
0 for screen 1 in Bank 5
1 for screen 2 in Bank 7

R0 Select the ROM

DS If set disables RAM paging so the extra memory cannot be used in 48K mode. Once set it cannot be disabled until the machine is reset

Figure 44.2: Memory/ROM paging port on 128K and +2

44.4 Spectrum +2A and +3 Memory Map

This section pertains to the +2A and +3 machines.

The Spectrum 128K **and** Spectrum +2 **machines use a different memory layout which is described in Section Section 44.3.**

The memory layout of the Spectrum +2A and Spectrum +3 machines is similar to that of the Spectrum 128K and Spectrum +2, except that it uses two ports to manage the paging, and four ROMs along with additional memory configuration options.

Here the two ports controlling the paging are located at `0x1FFD` and `0x7FFD`.

Bit 0 of port `0x1FFD` selects the paging mode. If 0 then it's in normal mode or 1 for an alternate memory map.

Table 44.2: Spectrum +2A/+3 Paging Ports

Port	RAM Copy
1FFD	5B67
7FFD	5B5C

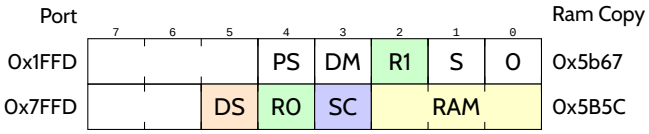
44.4.1 Normal Mode

When Bit 0 of port `0x1FFD` is clear then the machine is in normal mode. In this mode the RAM bank switching is like that on the Spectrum 128K, whilst the ROM switching selects one of the four available roms.

Figure 44.3 on page 852 shows the structure of these ports.

Table 44.3: Spectrum +2A/+3 ROM ID's

ID	ROM
0	128K Editor, Menu system, Self test
1	128K Syntax Checker
2	+3DOS
3	48K Basic



RAM The RAM bank to page into 0xC000…0xFFFF

SC The screen to display
0 for screen 1 in Bank 5
1 for screen 2 in Bank 7

R0 R1 Select the ROM.
Bit spans both ports

DS If set disables RAM paging so the extra memory cannot be used in 48K mode. Once set it cannot be disabled until the machine is reset

PS Printer Strobe, controls the printer port

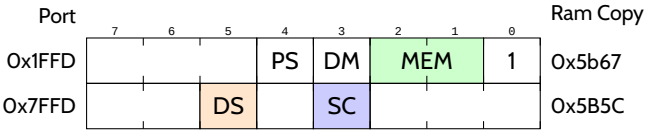
DM Controls the Disk Motor
1=motor on, 0=motor off

Figure 44.3: Memory/ROM paging port on Spectrum +2A and +3 in Normal mode

44.4.2 Special Mode

When Bit 0 of port `0x1FFD` is set then the machine is in special mode. In this mode, the RAM bits are ignored and bits 1 and 2 of `0x1FFD` control a memory map layout.

Figure 44.4 on page 853 shows the configuration of these ports when in Special Mode whilst Figure 44.5 on the next page shows the memory layouts available in this mode.



RAM The memory map mode to use

SC The screen to display
0 for screen 1 in Bank 5
1 for screen 2 in Bank 7

R0 R1 Select the ROM.
Bit spans both ports

DS If set disables RAM paging so the extra memory cannot be used in 48K mode. Once set it cannot be disabled until the machine is reset

PS Printer Strobe, controls the printer port

DM Controls the Disk Motor
1=motor on, 0=motor off

Figure 44.4: Memory/ROM paging port on +2A and +3 in Special mode

44.4.3 Plus 3 specifics

In normal operation, +3DOS on the Spectrum +3 uses RAM banks 1, 3, 4 and 6 for the disk cache and RAM disk. Bank 7 is used for the editor scratch pad and +3DOS workspace.

Address	Memory Mode			
	0	1	2	3
0000	Bank 0	Bank 4	Bank 4	Bank 4
4000	Bank 1	Bank 5	Bank 5	Bank 7
8000	Bank 2	Bank 6	Bank 6	Bank 6
C000	Bank 3	Bank 7	Bank 3	Bank 3

Figure 44.5: Spectrum Memory on +2A or +3 in Special Mode

45 File Formats

45.1 TAP Format

The .TAP file format is a container used to store data in a format identical to the Spectrum's SAVE ROM routine.

.TAP files can be concatenated together so forming a new tape is as easy as joining multiple .TAP files together to form a new one.

For example on Linux or MacOS:

```
cat part1.tap part2.tap > final.tap
```

In DOS and Windows:

```
copy /b part1.tap + part2.tap final.tap
```

At a bare minimum, a .TAP file consists of two blocks; a header block followed by a data block. Each block consists of the Block Wrapper which then contains data within it.

45.1.1 Block Wrapper

Header and Data blocks consists of a 2-byte word with the length of the block (including the checksum), the block type, then the block's data and finishing with a single checksum byte.

The block type is either 0x00 for a Header and 0xFF for a Data block.

The checksum byte is in reality a binary XOR of all bytes in the block and not a true checksum.

Byte	Description
0	Length of Block
1	
2	Block type
3	File Type
4	File Name padded with spaces
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	Data block length
15	
16	Parameter 1
17	Name for Type 1 & 2
18	Parameter 2
19	
20	Checksum

Figure 45.1: TAP Format Header Block

46 Machine Code

Page intentionally blank

Modern Day

Table of contents

47 Reference Files **861**
47.1 Available XML files . . . 861
47.2 Character Sets . . . 863
47.3 Processor instructions . . . 865
47.4 Screen Resolutions . . . 868

48 Reference API Files **871**
48.1 Acorn MOS API's . . . 871
48.2 Acorn MOS OSBYTE . . . 874
48.3 Acorn MOS OSWORD . . . 875
48.4 Spectrum 48K ROM API's . . . 877

49 Reference XML Format **879**
49.1 char - Define a character . . . 879
49.2 font - Define a bitmap font . . . 881
49.3 fontFile - Define a bitmap font source . . . 883
49.4 format - Define a machinecode instruction format . . . 884
49.5 formatType - Define a bit type used within a format . . . 887
49.6 generate - Define the content to generate . . . 890
49.7 hexgrid - Generate Hex Grids . . . 892
49.8 opcode - Define a dataset entry . . . 899
49.9 opcodeFigure - Generate tables for each entry . . . 901
49.10 opcodeFormat - Generate instruction formats . . . 903
49.11 opcodeList - Generate table of all entries . . . 906
49.12 opcodeFigure and opcodeList common features . . . 907
49.13 proc - root element . . . 918

47 Reference Files

The xml files located within the reference directory in the repository define multiple datasets ranging from processor instruction sets, character sets, fonts and operating system calls for various microcomputer systems.

Originally designed to aid in removing duplication and errors whilst documenting the 6502 and Z80 microprocessor instruction sets, the format has proved so useful that it's been reused for the other types of data.

The use of xml files allows for multiple types of content to be generated from that data, so that for most items, only a single source copy exists. This data can also be reused, not just within this project but with others using industry standard tools like XSLT.

47.1 Available XML files

The XML files within the repository are located in two directories.

47.1.1 Fonts

The font definitions are in the `reference/fonts` directory and are compiled first, allowing the later reference files to access them.

47.1.2 Datasets

The main datasets are in the `reference` directory.

acorn-mos Defines the public API entry points for the BBC Microcomputers MOS OS.

acorn-mos-osbyte Defines the OSBYTE calls within MOS.

character-sets The various character sets from ASCII to the ZX-81 and CBM PET *etc.* which use a non-ASCII character set.

processor-6502 The 6502, 65C02 and 65816 instruction sets

processor-z80 The Z80 instruction set

screen-resolutions Text and Graphics resolutions for multiple systems

Most are described here, but the API's are listed in Chapter 48 on page 871.

47.2 Character Sets

The `character-sets.xml` file stores the various character sets used by systems in a single dataset.

The dataset is keyed by the **hex** attribute.

Table 47.1 on the next page lists the fields in this dataset.

All the fields, other than **hex**, must be TEX safe, so where necessary use the appropriate TEX macro for a character.

For some character sets where we have an associated bitmap font, then use that font's macro to represent the character. That way the generated tables will use the actual characters for those machines, rather than a modern representation which usually will not match.

Table 47.1: List of fields in character-sets.xml

Field	Type	Content
hex	hexadecimal	The character code
op	name	The ASCII mnemonic of the character
ascii	string	The name of the character in ASCII
bbc	string	The BBC Microcomputer
spectrum	string	Character name on the Spectrum
zx81	string	Character name on the ZX-81

47.2.1 hex

As you would expect, this is the characters value in hexadecimal.

For ASCII this would be broken into three groups: `0x00`…`1F` for control codes, `0x20`…`7F` for the ASCII character set and `0x80`…`FF` for non ASCII characters.

Not all character sets map to ASCII.

47.2.2 op

This should be the ASCII mnemonic of the character or the actual character. Do not expand the mnemonic as thats the purpose of the **ascii** field.

For characters outside the ASCII range, *e.g. hex* $\geq 0x80$ this should be blank or omitted.

47.2.3 ascii

This describes the character from the ASCII character set. This is either the full name of the character, or the actual character itself.

For characters outside the ASCII range, *e.g. hex* $\geq 0x80$ this should be blank or omitted.

47.2.4 bbc

The BBC Microcomputer used ASCII except for the control codes which it had its own definitions, which are represented here. This character set also applied only for modes 0 ... 7 as mode 7 used the teletext character set for everything other than the control codes.

47.2.5 spectrum

The basic tokens were included in the Spectrum and ZX-81 manuals for their tables, hence why we include them here to reproduce them as close as possible.

The Spectrum was, unlike its predecessors, used ASCII. It also had the BASIC tokens mapped so those tokens are included.

47.2.6 zx81

The ZX-81 and its predecessor the ZX-80 had their own character set, so characters do not map with ASCII. It also had the BASIC tokens mapped so those tokens are included.

47.3 Processor instructions

The original purpose for the xml files was to ensure errors did not appear in the instruction lists for the 6502 and Z80. By making the xml a core source of information, repeated sections like tables and certain graphics would be generated from the xml ensuring that typing mistakes or omissions did not occur.

Here we have the two files `processor-6502.xml` and `processor-z80.xml`, each handling a processor family.

47.3.1 6502

The dataset, as defined by the **opcode**[1] element consists of the fields defined in Table 47.2.

1: **opcode** Section 49.8 on page 899

Table 47.2: List of fields in processor-6502 dataset

Field	Type	Content
hex	hexadecimal	Instruction byte value
op	name	Instruction name
mode	mode	Address mode
length	int	Length of instruction in bytes
cycles	int	Number of cycles to execute
proc	enum	Earliest processor supporting instruction

proc enum

The **proc** field determines the earliest processor which supported the instruction. When undefined it indicates the instruction has always been supported. The valid values are:

65C02 instruction was introduced on the 65C02

65816 instruction was introduced on the 65816

47.3.2 Z80

2: **opcode** Section 49.8 on page 899

The dataset, as defined by the **opcode**[2] element consists of the fields defined in Table 47.3.

Table 47.3: List of fields in processor-z80 dataset

Field	Type	Content
hex	hexadecimal	Instruction byte value
op	name	Instruction name
mode	mode	Address mode
length	int	Length of instruction in bytes
m	int	Number of cycles used by instruction
t	int	or cycles used when branch taken if m0/t0 defined
m0	int	If defined cycles used when branch not taken
t0	int	
group	string	Used to group some instructions
prefix	string	"opprefix" used as a style in main hex-grid
note	string	A comment, not used anywhere

m and t

The **m** and **t** attributes define the number of M-cycles and T-cycles the instruction takes to execute. Both are required to be present.

m0 and t0

These are identical to **m** and **t**, except that when present define the cycles taken when a branch or action does not take place. If used then both must be present.

e.g. the `DJNZ` instruction decrements B then branches if $B \neq 0$. If the branch took place then **m** and **t** apply, but if the branch did not occur then **m0** and **t0** define the number of cycles taken.

group

Some instructions are related but have different names. *e.g.* the block memory move instructions LDD, LDDR, LDI and LDIR are listed as four individual instructions although they are related and have the same instruction format. So, for those four they are grouped against the LDD instruction so it's instruction tables have all four listed together.

This also applies to the block in, out and compare instructions which follow the same pattern.

prefix

The **prefix** field is used to style the cells in the main hex-grid. Specifically, those cells which are a prefix and have their own hex-grid's are marked with "opprefix", so they appear with a yellow background.

47.4 Screen Resolutions

The `screen-resolutions.xml` file stores a dataset describing the screen resolutions of various systems.

The dataset, as defined by the **opcode**[3] element consists of the fields defined in Table 47.4.

3: **opcode** Section 49.8 on page 899

Table 47.4: List of fields in screen-resolutions.xml

Field	Type	Content
op	name	The system for this screen
mode	mode	The type of screen
textWidth	int	The number of characters
textHeight	int	
graphicsWidth	int	The number of pixels
graphicsHeight	int	
colours	int	The number of colours supported
colourNote	string	Note about the colours

47.4.1 mode

The **mode** field describes the type of screen. If undefined it indicates that the system has only a single mode available.

text

A common mode, used to describe a systems text only mode, usually the default one.

highres and medres

A common name for modes which are high or medium resolutions.

integer

When a system like the BBC Microcomputer uses an integer for its screen modes[4], then this should match those modes.

4: I do not list shadow modes as they are identical but with bit 7 set

47.4.2 colours

The **colours** attribute should state the maximum number of colours supported in this mode.

This is usually a number, and a power of two – but can be a textual value. *e.g.* the BBC Microcomputers Mode 7 is teletext so we state that here.

The optional **colourNote** attribute can be used to expand what this value means. *e.g.* the limitations of each character or pixel can have, but the main attribute is the maximum possible.

47.4.3 textWidth and textHeight

If both are present, the define the width and height of the screen in characters. If either are absent indicates this is a graphics-only mode.

47.4.4 graphicsWidth and graphicsHeight

If both are present, the define the width and height of the screen in pixels. If either are absent indicates this is a text-only mode.

Page intentionally blank

48 Reference API Files

Chapter 47 on page 861 defined the datasets related to specific processors or systems. However, another group of datasets describes API's. In this instance, these are the OS API's which programs can use on those systems. Due to how these are used, and the number of them, these have been split out from the rest.

48.1 Acorn MOS API's

The `acorn-mos.xml` file details the various API's within Acorns MOS OS.

Table 48.1 lists the fields in this dataset, whilst Listing 48.1 on page 873 shows an example of a single entry.

Table 48.1: List of fields in acorn-mos.xml

Field	Type	Content
hex	hexadecimal	Either addr or vAddr is addr is blank
addr op	hexadecimal name	Public entry point
vAddr vName	hexadecimal name	Vector of call if it exists
name	string	Description of call

48.1.1 hex attribute

The dataset is keyed by the **hex** attribute which represents the primary address for the api entry point. However, as MOS uses vectors for some entry points, then two other attributes are used, **addr** for the public address and **vAddr** for the vectors address. So for this dataset, **hex** is set to **addr** unless that does not exist, in which case **vAddr** should be used. This allows the set to remain ordered by address but allows us to show both sets of addresses in one go.

48.1.2 addr and op attributes

If the call has a public entry point then these attributes are used. **addr** is the address of the entry point whilst **op** the name for this entry point.

48.1.3 vAddr and vName attributes

If the call has a vector then these attributes are used. **vAddr** is the address of the vector, usually in page `0x02` whilst **vName** the name for the vector.

48.1.4 name attribute

This is a descriptive name for the entry point.

48.1.5 synopsis element

This defines the synopsis for each call. It should consist of a description about the call, the entry parameters, usually the registers involved, and the state when the call returns. Multiple **onExit** elements can be used if the call has differing return states.

Listing 48.1 on page 873 shows an example synopsis for the OSRDCH entry point. It has two exit states defined by **onExit** one for when the carry flag C is reset and one for when carry is set.

Listing 48.1: Example of api synopsis

```
<opcode hex="FFE0" addr="FFE0" op="OSRDCH" vAddr="0210" vName="RDCHV"
    name="Read character to A">
    <synopsis>
        <description>Read a character from the current input stream
    into A</description>
        <onExit title="C=0">
            <item key="A">\gls{ascii} character read</item>
            <item key="X, Y" preserved="true"/>
            <item key="N, V, Z" undefined="true"/>
        </onExit>
        <onExit title="C=1">
            <description>
                If an escape condition occurs then A=\proghexaddress{1B
    } and must be acknowledged with an \programCommand{OSBYTE} call
    with A=\proghexaddress{7E}.
            </description>
            <item key="A">Error number</item>
            <item key="X, Y" preserved="true"/>
            <item key="N, V, Z" undefined="true"/>
        </onExit>
    </synopsis>
</opcode>
```

48.2 Acorn MOS OSBYTE

The `acorn-mos-osbyte.xml` file details the various OSBYTE calls based on the A register on entry.

Table 48.2 on the following page lists the fields in this dataset, whilst Listing 48.2 on the next page shows an example of a single entry.

Table 48.2: List of fields in acorn-mos-osbyte.xml

Field	Type	Content
hex	hexadecimal	Either addr or vAddr is addr is blank
name	string	Description of call
bbcb	bool	BBC B only
bbcmaster	bool	BBC Master only
electron	bool	Acorn Electron only
var	string	Variables retrieved

48.2.1 hex attribute

As the OSBYTE call differenciates each action by the accumulator, this attribute is set to the required value to trigger the action.

48.2.2 name attribute

This is a descriptive name for the action.

48.2.3 bbcb attribute

Indicates the call is implemented on the BBC Model B

48.2.4 bbcmaster attribute

Indicates the call is implemented on the BBC Master 128

48.2.5 electron attribute

Indicates the call is implemented on the Acorn Electron.

48.2.6 var attribute

A comma delimited list of variables returned by this call. *e.g.* OSHWM or HIMEM

48.2.7 synopsis element

This defines the synopsis for each action. It should consist of a description about the call, the entry parameters, usually the X and Y index registers, and the state when the call returns. Multiple **onExit** elements can be used if the call has differing return states.

Listing 48.2 on the facing page shows an example synopsis for the OSRDCH entry point. It has two exit states defined by **onExit** one for when the carry flag C is reset and one for when carry is set.

Listing 48.2: Example of OSBYTE synopsis

```
1 <!-- to be defined -->
```

48.3 Acorn MOS OSWORD

The `acorn-mos-osword.xml` file details the various OSWORD calls based on the A register on entry.

Table 48.3 on the next page lists the fields in this dataset, whilst Listing 48.3 on the following page shows an example of a single entry.

Table 48.3: List of fields in acorn-mos-osword.xml

Field	Type	Content
hex	hexadecimal	Either addr or vAddr is addr is blank
name	string	Description of call

48.3.1 hex attribute

As the OSWORD call differenciates each action by the accumulator, this attribute is set to the required value to trigger the action.

48.3.2 name attribute

This is a descriptive name for the action.

48.3.3 synopsis element

This defines the synopsis for each action. It should consist of a description about the call, the entry parameters, usually the X and Y index registers, and the state when the call returns. Multiple **onExit** elements can be used if the call has differing return states.

Listing 48.3 on the next page shows an example synopsis for the OSRDCH entry point. It has two exit states defined by **onExit** one for when the carry flag C is reset and one for when carry is set.

Listing 48.3: Example of OSWORD synopsis

```
1 <!-- to be defined -->
```

48.4 Spectrum 48K ROM API's

The `spectrum-48.xml` file details the various API's within Sinclair's Spectrum ROM.

Table 48.4 on the facing page lists the fields in this dataset, whilst Listing 48.4 on the next page shows an example of a single entry.

Table 48.4: List of fields in acorn-mos.xml

Field	Type	Content
hex op	hexadecimal name	Public entry point
name alt	string string	Description of call alternate access method

48.4.1 hex attribute

The dataset is keyed by the **hex** attribute which represents the primary address for the api entry point. However, as MOS uses vectors for some entry points, then two other attributes are used, **addr** for the public address and **vAddr** for the vectors address. So for this dataset, **hex** is set to **addr** unless that does not exist, in which case **vAddr** should be used. This allows the set to remain ordered by address but allows us to show both sets of addresses in one go.

48.4.2 op attributes

The name for this entry point.

48.4.3 name attribute

This is a descriptive name for the entry point.

48.4.4 synopsis element

This defines the synopsis for each call. It should consist of a description about the call, the entry parameters, usually the registers involved, and the state when the call returns. Multiple **onExit** elements can be used if the call has differing return states.

Listing 48.4 on the following page shows an example synopsis for the OSRDCH entry point. It has two exit states defined by **onExit** one for when the carry flag C is reset and one for when carry is set.

Listing 48.4: Example of api entry

```
<opcode hex="0556" op="LDBYTES" name="Load bytes from cassette">
    <synopsis>
        <description>
            Loads a block of bytes from cassette.
        </description>
        <onEntry>
            <description>
                Due to how the code works, the D register cannot be \
    proghexaddress{FF} on entry.
            </description>
            <item key="A">\proghexaddress{00} for header or \
    proghexaddress{FF} for data block</item>
            <item key="Carry">Set for load, reset for verify</item>
            <item key="DE">Length of block to save</item>
            <item key="IX">Base address of block</item>
        </onEntry>
        <onExit>
            <item key="Carry">Clear on error</item>
        </onExit>
    </synopsis>
</opcode>
```

Reference XML Format 49

The following sections define each element. They are listed in alphabetical order, although the **proc**[1] element is the first one to start with as that's the XML document's root element.

1: **proc** Section 49.13 on page 918

49.1 char
Define a character

The **char** element declares a character using a hexadecimal string.

It can be used along with **fontFile**[2] where the character definitions provided by **char** will be applied after they are read from the source file.

2: **fontFile** Section 49.3 on page 883

The **char** element has a single attribute, **hex** which is the hex code of the char being defined.

The content of the **char** element contains the hexadecimal definition of the character. Only the characters $0 \ldots 9 A \ldots F$ are valid. White space is ignored.

Listing 49.1 on the next page shows an example taken from the Spectrum Font. It shows the reading font data from a ROM image and then adding some characters not present in the image.

The ROM image contains characters `0x00`…`7F` starting from position address `0x3D00` in that image.

We then supply the missing block characters `0x80`…`8F`.

The result of this are the following characters defined in the **char** elements.



Characters `0x86`, `0x8C` and `0x89`

Listing 49.1: Example of char element

```
<processor>
    <generate dir="figs/font">
        <font filename="spectrum.tex"
            csname="zxspectrum"
            title="ZX Spectrum Font">

            <!--
                Take bitmap from opense.rom which
                is part of the fuse emulator
            -->
            <fontFile filename="opense.rom"
                start="20" end="7F" offset="3D00"/>

            <!-- Additional block characters -->
            <char hex="80">00000000 00000000</char>
            <char hex="81">0F0F0F0F 00000000</char>
            <char hex="82">F0F0F0F0 00000000</char>
            <char hex="83">FFFFFFFF 00000000</char>
            <char hex="84">00000000 0F0F0F0F</char>
            <char hex="85">0F0F0F0F 0F0F0F0F</char>
            <char hex="86">F0F0F0F0 0F0F0F0F</char>
            <char hex="87">FFFFFFFF 0F0F0F0F</char>
            <char hex="88">00000000 F0F0F0F0</char>
            <char hex="89">0F0F0F0F F0F0F0F0</char>
            <char hex="8A">F0F0F0F0 F0F0F0F0</char>
            <char hex="8B">FFFFFFFF F0F0F0F0</char>
            <char hex="8C">00000000 FFFFFFFF</char>
            <char hex="8D">0F0F0F0F FFFFFFFF</char>
            <char hex="8E">F0F0F0F0 FFFFFFFF</char>
            <char hex="8F">FFFFFFFF FFFFFFFF</char>
        </font>
    </generate>
</processor>
```

49.2 font
Define a bitmap font

The **font** element generates a LaTeX compatible file which can generate an exact replica of an 8-bit font. It does this by using original or modern ROM images containing the actual bitmaps used by machines for their fonts.



Note

These are not TeX fonts, but define a macro which can generate an individual character when needed.

49.2.1 Attributes

filename

The filename of the TeX file to generate.

csname

The TeX csname of the macro to generate.

title

The title used in Volume 0's figure index

49.2.2 Child elements

The **font** can contain a single optional **fontFile**[3] which defines a source of raw bitmap data, usually a ROM image.

It can also contain multiple **char**[4] elements which can define a character from hexadecimal within the xml file.

3: **fontFile** Section 49.3 on page 883

4: **char** Section 49.1 on page 879

49.2.3 Example font

Listing 49.2 on the following page shows the Apple-II font. **fontFile** defines the `apple-ii.tex` file using the csname `appleii`. **fontFile**[5] defines the bitmap are in the file `apple-ii.pf`.

The **hexgrid**[6] element generates a grid showing the entire font[7]

5: **fontFile** Section 49.3 on page 883

6: **hexgrid** Section 49.7 on page 892

7: As this font only defined characters `0x20…7F` we don't generate the right hand page, which is commented out here.

49.2.4 Using a font

To use the font simply include it in the document and pass the hexadecimal value to the macro with the csname.

e.g. with the AppleII font to get the * character then we would use `\appleii{2A}` to get ⁕ .

Using the Apple II font
```
\input{figs/font/apple-ii}
\appleii{2A}
```

Listing 49.2: Example of font element

```
<proc>
    <generate dir="figs/font">
        <font filename="apple-ii.tex" csname="appleii" title="Apple II Font">
            <fontFile filename="apple-ii.pf" start="20"/>
        </font>

        <hexgrid ascii="true">
            <pattern src="$(hex)">^([0-9A-F])([0-9A-F])$</pattern>
            <file fileName="apple-ii-verso.tex" persist="true"/>
            <title>Apple II Character Set</title>
            <indexVol0>Font!Apple II</indexVol0>
            <xaxis>$$(hx)\times$</xaxis>
            <yaxis>$\times $(hy)$</yaxis>
            <default>\path ($(nodeId)) node[opcode] {\appleii{$(hex)}} node[addrMode] {$(basicToken)} node[topLeft]
                {$(toDec|hex)} node[topRight] {$(hex)};
            </default>
        </hexgrid>

    </generate>

</proc>
```

49.3 fontFile
Define a bitmap font source

The **fontFile** element declares that a font is to get its bitmap data from a file, usually a ROM image.

Listing 49.1 on page 880 has an example of the **fontFile** in use.

49.3.1 Attributes

The following attributes are available:

filename

The filename of the source file.

offset

The offset, in hexadecimal, within the source file where the data begins. If not defined then 0 is used.

start

The start character number, in hexadecimal. If not defined then 00 is used.

end

The end character number, in hexadecimal. This is inclusive, so if end is set to `7F` then the last character read will be `0x7F`. If not defined then the end of the source file will mark the end point[8].

8: In reality the character $2^{63} - 1$ is the last character

49.4 format
Define a machinecode instruction format

The **format** element defines the possible bit formats for a Machine Code instruction.

The **opcodeFormat**[9] element uses these and the related **formatType**[10] elements to generate both tables and graphical representations of these formats.

9: opcodeFormat
Section 49.10 on page 903

10: formatType
Section 49.5 on page 887

49.4.1 op attribute

The element contains a single attribute **op** which holds the instruction name this format defines.

49.4.2 description element

The element then contains a **description**[11] element which contains what the format describes.

11: I use this in the xml but the processor does not know about it - it's essentially documentation only at this point.

49.4.3 type element

It then contains one or more **type** elements describing each possible format.

Listing 49.3 on page 886 shows an example of this for the ADC instruction for the Z80.

49.4.4 type.label attribute

The **type** element consists of the **label** attribute which describes the type.

49.4.5 type.byte element

It then has one or more **byte** elements within it, each describing each byte of the instruction.

The body consists of 8 characters[12], one for each bit.

The values are either a '0' or '1' for a bit that is permanently set or reset, or a letter defining bits which form part of the instructions' definition.

For example, line 4 in Listing 49.3 on page 886 shows `10001rrr`. Here the `rrr` refers to three bits which use the **formatType**[13] using the label "r".

Line 8 shows where an entire byte is used for a number, in this case type "n".

Line 11 shows a constant, where all bits are either '0' or '1'.

12: Currently this is a single byte but might be expanded later for 16-bit machines

13: formatType
Section 49.5 on page 887

49.4.6 type.byte.label attribute

Any byte can have a **label** attached which is then included in the generated graphics. The label is used to identify the byte.

When the byte is a constant, *e.g.* it only consists of 0 or 1's, the **label** is set with the hexadecimal value of the byte, making it easy to the reader to decode the bit pattern.

Another use of **label** is when a value is split over two bytes.

For example line 19 shows the Z80 `CALL` instruction which takes a 16-bit address. As that address is split over two bytes, the bytes content consists of the formatType 'n' but we set each ones labels to "low byte" and the other the "high byte". To the reader they then know which one takes which half of the address.

Listing 49.3: Example of defining an instruction format

```
<format op="ADC">
    <description>Add with Carry</description>
    <type label="A,r">
        <byte>10001rrr</byte>
    </type>
    <type label="A,n">
        <byte label="CE">11001110</byte>
        <byte>nnnnnnnn</byte>
    </type>
    <type label="A,(HL)">
        <byte label="8E">10001110</byte>
    </type>
</format>

<format op="CALL">
    <description>Call Subroutine</description>
    <type label="nn">
        <byte label="CD">11001101</byte>
        <byte label="low byte">aaaaaaaa</byte>
        <byte label="high byte">aaaaaaaa</byte>
    </type>
</format>

<format op="LD">
    <description>Load Register</description>
    <type label="A,(BC)">
        <byte label="0A">00001010</byte>
    </type>
    <type label="r,r'">
        <byte>01rrrRRR</byte>
    </type>
</format>
```

49.5 formatType Define a bit type used within a format

The **formatType** element defines the possible bit formats used within a **format**[14].

The **opcodeFormat**[15] element uses these and the related **format**[16] elements to generate both tables and graphical representations of these formats.

The element consists of a few attributes and two child elements.

14: **format** Section 49.4 on page 884

15: **opcodeFormat** Section 49.10 on page 903

16: **format** Section 49.4 on page 884

49.5.1 caption attribute

The **caption** attribute describes the format type and is used as a caption when the type is included in the output.

49.5.2 label and text attributes

The **label** attribute defines the label for the type. This is a single character, either a lower or upper case letter, which is used within the bit patterns within a **format**.

The optional **text** attribute provides the text to show for the label. If this is not provided then the label's value is used.

49.5.3 altLabel and altText attributes

Sometimes n instruction refers to two entries using the same format type.

e.g. the Z80 has instructions which reads from one register and stores in another. In this case we can declare an alternate label with **altLabel** which will also use the same format.

Similar to the normal **label**, the **altText** attribute defines the output text for the label, defaulting to the value of **altLabel** if it's not defined.

Line 9 of Listing 49.4 on page 889 shows an example of this. Here we set the alternate label `R` which we can then use in a format definition. However, we set the text for this to be `r'`.

Line 30 in Listing 49.3 on page 886 shows the use of both **label** and **altLabel** being used within a format definition.

49.5.4 style element

The **style** element provides the TikZ style to use when including this type in a format. Line 2 of Listing 49.4 on the facing page shows an example of this where the text colour is black, the draw colour (*e.g.* lines, borders *etc.*) is also black and the fill is the NoteGoldenRod[17] colour.

17: NoteGoldenRod

49.5.5 byte element

The **byte** element declares a bit format within the type. The **label** attribute contains the bits set, whilst the elements body contains TEX that describes the value.

e.g. Line 3 shows that "00" refers to the BC register pair, whilst "11" the Stack Pointer.

Line 22 shows where using TEX here is needed. This is used for the Z80 `RST` instruction so we show both the hexadecimal value and then, right justified the decimal value for each option.

Not every combination of bits needs to be defined. In fact, in some cases a combination may be excluded. For the `r` type defined in line 9 the bit combination "110" is notably absent.

For the Z80 that bit combination does not get used for a register, but usually used by another instruction. *e.g.* for some it refers to `(HL)`, others the flag register `F` and on one instance an unrelated instruction[18].

18: HALT if you are wondering, replacing the non-sensical `LD (HL),(HL)` instruction

Listing 49.4: Example of defining a formatType

```
<formatType label="s" caption="Register Pair">
    <style>color=black,draw=black,fill=NoteGoldenRod</style>
    <byte label="00">BC</byte>
    <byte label="01">DE</byte>
    <byte label="10">HL</byte>
    <byte label="11">SP</byte>
</formatType>

<formatType label="r" caption="Register" altLabel="R" altText="r'">
    <style>color=black,draw=black,fill=NoteGoldenRod</style>
    <byte label="111">A</byte>
    <byte label="000">B</byte>
    <byte label="001">C</byte>
    <byte label="010">D</byte>
    <byte label="011">E</byte>
    <byte label="100">H</byte>
    <byte label="101">L</byte>
</formatType>

<formatType label="p" caption="Restart">
    <style>color=black,draw=black,fill=green!50</style>
    <byte label="000">\proghexaddress{00}\hfill0</byte>
    <byte label="001">\proghexaddress{08}\hfill8</byte>
    <byte label="010">\proghexaddress{10}\hfill16</byte>
    <byte label="011">\proghexaddress{18}\hfill24</byte>
    <byte label="100">\proghexaddress{20}\hfill32</byte>
    <byte label="101">\proghexaddress{28}\hfill40</byte>
    <byte label="110">\proghexaddress{30}\hfill48</byte>
    <byte label="111">\proghexaddress{38}\hfill56</byte>
</formatType>
```

49.6 generate
Define the content to generate

The **generate** element defines the content to generate at build time. The files generated can be stored within the git repository so that the content does not need to be build from scratch every time – especially if the tool cannot be run on someone's local environment.

When the content is generated, the first line contains the timestamp the file was generated. To prevent needless commits where just the timestamp has changed, it will only write the file if the rest of the file has changed, ignoring the first line.

49.6.1 dir attribute

There is only a single attribute called **dir** which defines the target directory for all content within this element. If a separate directory is required then a second **generate** element should be used.

49.6.2 Child elements

The **generate** can contain any of the elements shown in Table 49.1 on page 890. All of them can be repeated as many times as required.

Table 49.1: List of child elements for generate

Element	Description	Section	
font	Defines a bitmap font	49.2	881
hexgrid	Defines a hex-grid of the dataset	49.7	892
opcodeList	Generate tables for entire set	49.11	906
opcodeFigure	Generate tables for individual entries	49.9	901
opcodeFormat	Generate instruction formats	49.10	903
synopsys	Generate synopsis entries	??	??

Listing 49.5 on the facing page shows an example for the 6502. All content generated will be written into the `figs/6502/gen` directory in the repository.

Listing 49.5: Example of generate element

```
<!-- Generate content for the 6502 -->
<proc processor="6502">
  <!-- place generated content into the figs/6502/gen directory -->
  <generate dir="figs/6502/gen">

    <!-- Hex Grid of 6502 instructions -->
    <hexgrid pattern="^([0-9A-F])([0-9A-F])$">
      <file fileName="hexgrid-verso.tex" persist="true"/>
      <file fileName="hexgrid-recto.tex" persist="true" right="true"/>
      <title>6502 Opcodes</title>
      <indexVol0>6502!HexGrid!$(opName|toUpper)</indexVol0>
      <start>65C02/.style={fill=yellow}, 65816/.style={fill=green!50},</start>
      <xaxis>$\times $(hx)$</xaxis>
      <yaxis>$$(hy)\times$</yaxis>
      <style>$(proc)</style>
      <default>\path ($(nodeId)) node[opcode] {$(upper|op)} node[addrMode] {$(mode)} node[topLeft] {$(hex)} node[topRight] {$(length)};</default>
    </hexgrid>

    <!-- Longtable of opcodes ordered by op -->
    <opcodeList file="by-opcode.tex" preview="true">
      <caption>6502 Instructions by Opcode</caption>
      <indexVol0>6502!Instructions!by Opcode</indexVol0>
      <label>asm:6502:rep:opcode</label>
      <orderBy>
        <field>op</field>
        <field>hex</field>
      </orderBy>
      <field name="opcode" title="Opcode" style="typewriter"/>
      <field name="hex" title="Hex" style="typewriter"/>
      <field name="length" title="Length" rotateHeader="90" align="r"/>
      <field name="cycles" title="Cycles" rotateHeader="90" align="r"/>
    </opcodeList>

  </generate>
</proc>
```

49.7 hexgrid
Generate Hex Grids

The **hexgrid** element generates a hex-grid from the dataset. A hex-grid is a two-dimensional matrix of the dataset using the **hex** attribute as the key.

Each A5 page can contain 128 characters, so hex-grids come in two flavours, left (default) for verso pages and right for recto pages – where the y-axis is generated on the respected outside edge of the grid. In normal use the verso side has hex values `0x00`…`7F` whilst the recto side `0x80`…`FF`.

Each axis takes its coordinate from each nibble of the lower byte of that value.

For normal grids the x-axis is the lower nibble whilst the y-axis is the upper nibble. This means the left/verso page has values $x0 \ldots x7$ whilst the right page values $x8 \ldots xF$.

However, some grids like ASCII style tables have them reversed and this is supported with the **ascii** attribute. For those style of tables, the pages have values $00 \ldots 7F$ on the left/verso page whilst the right/recto page values $80 \ldots FF$.

This also has the benefit that, for those like ASCII which are 7-bit, we only need a single page to hold the entire character set rather than two with half of those pages being blank.

To support grids with more than 256 possible entries, a prefix can be applied so that the grid only includes those entries who's hex value has that prefix. The Z80 instruction set uses this where there are several prefix bytes for some instructions.

Listing 49.6 on page 898 shows an example of **hexgrid** in use.

49.7.1 ascii attribute

If set to "true" then the x and y axes are flipped, so the x-axis has the upper nibble whilst the y-axis the lower nibble. This is used commonly for character sets, hence it's name, as they tend to be organized this way.

It also allows for 7-bit character sets like ASCII to be presentable on a single page rather than across two pages with half of the grid being blank.

49.7.2 escape attribute

If defined, determines if the content of fields will be escaped to make then TEX safe. If undefined then escaping will occur.

This should be done with caution in case the content of the dataset breaks the document, but is used when some entries like ASCII tables need to use macros in the dataset.

49.7.3 pattern attribute

The **pattern** attribute defines a regular expression which matches the **hex** attribute in **opcode**[19]. It should have two capture groups in it, the first for the upper and the second for the lower nibble of the opcode.

19: **opcode** Section 49.8 on page 899

For more complex tables like the Z80 instruction set which has prefixes, the pattern can match those prefixes so we have tables specific to each capture group.

```
^([0-9A-F])([0-9A-F])$
```

The default pattern[20] which captures all hex values between `0x00`...`FF`. This will be used if a **pattern** attribute is not provided.

20: Always match for uppercase characters as the **hex** field is always converted into uppercase.

```
^CB([0-9A-F])([0-9A-F])$
```

An example pattern which matches a prefix, in this instance `CB` then two more hexadecimal digits which are used by the hex-grid. In this example this would include Z80 instructions whose prefix byte is `0xCB`.

49.7.4 file element

The **file** child element defines the output filename to generate. It also defines if the grid is for the right/recto side, defaulting to the left/verso side. This setting also defines which half of the 256 range for the grid is rendered.

There has to be at least one of these defined, otherwise no output will be created.

In line 4 in Listing 49.6 on page 898 we declare the verso or left page of the grid. We include verso in the filename, but it's not necessary.

Line 5 shows the recto or right page of the grid. Note, some grids like ASCII tables do not have a recto page.

49.7.5 match element

The **match** child element define rules to select which style of cell is to be rendered if they match. If no match is found then the **default** style is used.

An example of this would be if you had a column with a set of possible values, and you wanted to use a specific node pattern for that specific cell. Here you would define a **match** with the **field** and **value** attributes set appropriately. When that field has that value then the contents of the **match** element is used for the node style.

This deals with custom formatting of the cell. For a more generic style, *e.g.* custom colours then that's the **style** attribute.

49.7.6 xaxis and yaxis elements

The **xaxis** and **yaxis** elements define the labels for their respective axis. This usually takes one of the variables shown in Table 49.2 on page 897.

In the example in Listing 49.6 on page 898, in line 11 we use math mode to typeset the x-axis label. Line 12 handles the y-axis.

49.7.7 style element

The **style** element allows for a style to be defined by a field. This style can then be defined with the **start** element to provide custom colours dependent on the type of cell. This is usually used more often than using the **match** element for customizations.

An example of this is the 6502 instruction sets where the *proc* field defines the type of processor the instruction is available on.

e.g. Listing 49.6 on page 898 shows an example where the **style** element on line 18 will get the style from the *proc* field for that entry. TikZ will then use that for the style name – in this case "65C02" or "65816" are valid style names.

In that example, an instruction available since the 65c02 is yellow and those since the 65816 are filled in green.

The **start** element in line 15 defines the possible styles.

49.7.8 default element

The **default** element is required and defines the format of each cell. This is pure TikZ so is usually a **\path** or **\node** statement.

In line 21 of Listing 49.6 on page 898 we have the default cell format defined.

This breaks down into a **\path** terminated with a semi-colon ';'

```
1 \path
2     ($(nodeId))
3     node[opcode] {$(upper|op)}
4     node[addrMode] {$(mode)}
5     node[topLeft] {$(hex)}
6     node[topRight] {$(length)}
7 ;
```

1. **Required**, the **\path** command in TikZ.
2. **Required**, anchors the path to the cell within the grid so everything is relative to it.
3. Style `[opcode]` to show the field `op` in uppercase at the center of the cell.
4. The style `[addrMode]` to show the field `mode`, below the op name.
5. In the `[topLeft]` corner in a small font the `hex` field
6. In the `[topRight]` corner in a small font the `length` field
7. **Required**, terminates the **\path** command in TikZ.

A common alternative for line 6, used by ASCII tables, is to have the character code in decimal in the top-right corner whilst the hex code is on the top-left.

This is defined using the following:

```
1 node[topRight] {$(hex|toDec)}
```

49.7.9 legend element

The **legend** element is optional and allows for a legend to appear in the margin. Each legend entry is effectively a cell so is done using the same format as the **default** element, but customized to that specific styling be it with the **match** or **style** elements.

legend is similar to **default** in that it's content defines how the note is rendered. The main difference is that static content should be placed in there, as lines from 24 show. The **legend** on that line shows the default cell layout with the appropriate static content in place.

49.7.10 $(*variable*) expansion

Many elements and attributes accept variables via the **$(*variable*)** notation. Whenever this is seen, the *variable* portion is checked against Table 49.2 on the facing page and if not there then as a field name in the current row.

The value can then be transformed adding a transformation name after variable name. *e.g.* `hex|toDec` converts the hex field into decimal. Table 49.4 on page 917 shows the available transforms.

Table 49.2: Variables within hex-grid

Variable	Description
X Axis	
x	Column 0 … 7 on verso, 8 … 15 on recto
dx	Column 0 … 7 on either side
hx	X as a hexadecimal digit
Y Axis	
y	Row 0 … 15
hy	Y as a hexadecimal digit
Cells	
xy	Nodes decimal value 0 … 255
xyhex	xy as two digit hexadecimal value
nodeId	Unique node id for the current cell

Listing 49.6: Example of hexgrid

```
<!-- Hex Grid of 6502 instructions -->
<hexgrid pattern="^([0-9A-F])([0-9A-F])$">
    <!-- generate 2 files one each for left and right pages -->
    <file fileName="hexgrid-verso.tex" persist="true"/>
    <file fileName="hexgrid-recto.tex" persist="true" right="true"/>

    <title>6502 Opcodes</title>
    <indexVol0>6502!HexGrid!$(opName|toUpper)</indexVol0>

    <!-- labels for x-axis and y-axis -->
    <xaxis>$\times $(hx)$</xaxis>
    <yaxis>$$(hy)\times$</yaxis>

    <!-- define TikZ styles for decorating specific cells -->
    <start>65C02/.style={fill=yellow}, 65816/.style={fill=green!50},</
    start>

    <!-- select the style to use from the proc field -->
    <style>$(proc)</style>

    <!-- cell layout -->
    <default>\path ($(nodeId)) node[opcode] {$(upper|op)} node[addrMode
    ] {$(mode)} node[topLeft] {$(hex)} node[topRight] {$(length)};</
    default>

    <!-- legend for default cells, e.g. proc=="" -->
    <legend>\path ($(nodeId)) node[opcode] {6502} node[addrMode] {addr
    mode} node[topLeft] {hex} node[topRight] {len};</legend>

    <!-- legend for when proc=="65C02" -->
    <legend style="65C02">\path ($(nodeId)) node[opcode] {65C02} node[
    addrMode] {addr mode} node[topLeft] {hex} node[topRight] {len};</
    legend>

    <!-- legend for when proc=="65816" -->
    <legend style="65816">\path ($(nodeId)) node[opcode] {65816} node[
    addrMode] {addr mode} node[topLeft] {hex} node[topRight] {len};</
    legend>
</hexgrid>
```

49.8 opcode
Define a dataset entry

The **opcode** element defines a row or entry within the dataset. It was originally meant to represent a Machine Code instruction, hence it's name, but is now used to define an entry.

Each entry consists of a group of pre-defined attributes and then custom attributes specific to the data set. Each entry can also contain some child elements if the dataset requires them.

49.8.1 hex attribute

The unique hexadecimal id for this entry. Originally used for the Machine Code instruction, this is now used for identifying the content being defined. *e.g.* for the MOS OSBYTE calls, this is the value of the A register defining the call.

If this is not present it is automatically generated.

If multiple entries have the same value then the build will fail with an error as this must be unique.

49.8.2 op attribute

Originally used to define a Machine Code instructions name, this is now a generic name for the entry. Along with the **group** attribute, this can be used to group instructions or entries together.

49.8.3 mode attribute

Used for Machine Code instructions, this defines the address mode for the instruction. The mode[21] element can be used to define details about the mode if needed.

21: mode **??**

49.8.4 opCode attribute

This is used as the instruction's name which is usually a combination of the **op** and **mode** fields. As such it is optional and defaults to be "**op mode**".

49.8.5 group attribute

If defined this will group this entry with a parent entry having its op attribute matching this one. Mainly used by Z80 instructions like LDDR, LDI and LDIR which are grouped with LDD so they share the same format[22].

22: format Section 49.4 on page 884

49.8.6 Custom Attributes

The predefined attributes are fine for most uses but to allow for more expansive datasets, any attribute which is not part of the predefined list are placed in a list, so they can be referenced by name within the generate[23] element.

23: generate Section 49.6 on page 890

49.8.7 synopsis element

Currently, the **opcode** element can contain just a synopsis[24] element.

24: synopsis ??

49.9 opcodeFigure Generate tables for each entry

The **opcodeFigure** element generates figures from datasets. It takes the same attributes as **opcodeList**[25].

The main difference, however, is that here it will group all entries by their **op** field generating a table for each one in its own file.

25: **opcodeList** Section 49.11 on page 906

49.9.1 filename attribute

To facilitate this, the filename in the elements **filename** attribute will have the characters %% replaced by the upper case value of the **op** field.

49.9.2 Example

Listing 49.7 on the next page shows an example for the Z80.

See Section 49.12 on page 907 for the rest of the functionality.

In this example:

%% here will be expanded to the op field just like the filename.

- 2 Declares we want the table to be a TEX longtable, not to include a caption. The **prefiew** attribute indicates we want to include it in full in Volume 0's figure index. If this was false or undefined then only an entry would exist but not the content. This is important for TikZ content which will be described later.
- 3 of the table, but unused here as we have **noCaption** set.
- 4 is the index entry used in Volume 0.
- 5 is the label for the table.
- 6 defines how the entries of the table is to be ordered
- 11 Declares the fields included in the table, from left to right.

Listing 49.7: Example of opcodeFigure

```
<!-- Table per opcode -->
<opcodeFigure file="opcode/$(opName|toUpper).tex" noCaption="true"
    tableType="long" preview="true">
    <caption>Z80 Instruction $(opName|toUpper)</caption>
    <indexVol0>Z80!Instruction!$(opName|toUpper)</indexVol0>
    <label>asm:z80:rep:$(opName|toUpper)</label>
    <orderBy>
        <field>op</field>
        <field>mode</field>
        <field>hex</field>
    </orderBy>
    <field name="opcode" title="Opcode" style="typewriter"/>>
    <field name="hex" title="Hex" style="typewriter" transform="bytes"/
    >
    <field name="length" title="Length" rotateHeader="90" align="r"/>
    <field name="m" title="M cycles" rotateHeader="90" align="r"/>
    <field name="t" title="T states" rotateHeader="90" align="r"/>
    <field name="m0" title="M not taken" rotateHeader="90" align="r"/>
    <field name="t0" title="T not taken" rotateHeader="90" align="r"/>
</opcodeFigure>
```

Listing 49.8: Example of opcodeList

```
<!-- Longtable of opcodes ordered by op -->
<opcodeList file="by-opcode.tex" preview="true">
    <section>Instructions by Opcode</section>
    <caption>Z80 Instructions by Opcode</caption>
    <indexVol0>Z80!Instructions!by Opcode</indexVol0>
    <label>asm:z80:rep:opcode</label>
    <orderBy>
        <field>op</field>
        <field>mode</field>
        <field>hex</field>
    </orderBy>
    <field name="opcode" title="Opcode" style="typewriter"/>
    <field name="hex" title="Hex" style="typewriter" transform="bytes"/
    >
    <field name="length" title="Length" rotateHeader="90" align="r"/>
    <field name="m" title="M cycles" rotateHeader="90" align="r"/>
    <field name="t" title="T states" rotateHeader="90" align="r"/>
</opcodeList>
```

49.10 opcodeFormat Generate instruction formats

The **opcodeFormat** element generates TikZ pictures of instructions based on the **format**[26] and **formatType**[27] elements.

26: **format** Section 49.4 on page 884

27: **formatType** Section 49.5 on page 887

49.10.1 file attribute

This is required and defines the filename for the generated content. As this generates multiple files, the %% pattern is used and is substituted with the **op** attribute from the **format** element.

49.10.2 margin attribute

If set then the entry in Volume 0's figure index will use the margin layout. By default, the figure index uses the wide layout but some figures need to use the margin layout.

An example is when generating the format on a per-instruction basis. As this generates not just a table but includes the **formatType**'s in the margins, we need to specify margin. Line 2 in Listing 49.9 on page 905 shows this in action.

49.10.3 caption element

Defines the caption of the table.

49.10.4 label element

Defines the label of the table.

49.10.5 index element

Defines the index entry in Volume 0 for the table. This is not used in the main document.

49.10.6 prefix element

A **opcodeFormat** can contain one or more **prefix** elements, each one defining the prefix to use when generating a decode table based on the provided formats.

Line 13in Listing 49.9 on page 905 shows this for the Z80.

The presence of a **prefix** element triggers the generation of the decode table.

For processors which have no prefixes, then an empty **prefix** element needs to be used to trigger the generation of the decode table.

However, when prefixes are present, then the empty prefix is always generated, so is optional.

Line 24 shows what this would look like for the 6502.

Listing 49.9: Example of generating formats

```
<!-- Table per opcode/format -->
<opcodeFormat file="format/$(opName|toUpper).tex" layout="margin">
    <caption>Z80 Instruction Format $(opName|toUpper)</caption>
    <indexVol0>Z80!Instruction!$(opName|toUpper)</indexVol0>
    <label>asm:z80:fmt:$(opName|toUpper)</label>
</opcodeFormat>

<!-- Table for all opcodes -->
<opcodeFormat file="format-$(opName|toUpper).tex">
    <caption>Z80 Instruction Decode Table $(opName|toUpper)</caption>
    <indexVol0>Z80!Decode Table!$(opName|toUpper)</indexVol0>
    <label>asm:z80:decode</label>
    <prefix>CB</prefix>
    <prefix>DD</prefix>
    <prefix>ED</prefix>
    <prefix>FD</prefix>
</opcodeFormat>

<!-- Example decode table for processor with no prefixes -->
<opcodeFormat file="format-$(opName|toUpper).tex">
    <caption>6502 Instruction Decode Table $(opName|toUpper)</caption>
    <indexVol0>6502!Decode Table!$(opName|toUpper)</indexVol0>
    <label>asm:6502:decode</label>
    <prefix/>
</opcodeFormat>
```

49.11 opcodeList
Generate table of all entries

The **opcodeList** element generates a table from a dataset.

49.11.1 filename attribute

28: opcodeFigure
Section 49.9 on page 901

It is similar to **opcodeFigure**[28] except that here it generates a single file for the entire dataset, whereas **opcodeFigure** generates a file for each **op**.

Listing 49.8 on page 902 shows an example of **opcodeList**.

49.11.2 Example

See Section 49.12 on the next page for the rest of the functionality.

49.12 opcodeFigure and opcodeList common features

If you are looking at the go source, this section is in the `opcodetable.go` file, separate from the other two

This section documents the common features of **opcodeFigure** and **opcodeList**.

49.12.1 group attribute

If this is set to "true" then the table is wrapped in a TEX group.

49.12.2 caption element and noCaption attribute

The **caption** element defines the caption of the table in the document and the relevant list of index. However, the **noCaption** attribute will, if set to "true" cause the caption not to be used.

49.12.3 allowEmpty

If "true" then this would generate the table even if it had no rows in it – *i.e.* just the table headers. The default is not to generate the table and hence the file.

49.12.4 layout attribute

This defines the page layout. It takes one of three values:

Defaults to "**margin**" if undefined.

margin	The default 2/1 margin layout
wide	The Wide portrait layout
landscape	A wide landscape layout where content is rotated 90 degrees counter-clockwise.

49.12.5 clearPage attribute

The **clearPage** attribute determines if a new page is created before each entry.

It takes one of the following values:

Defaults to "**false**" if undefined.

false A new page is never started.

true A new page is always started before each entry.

notFirst A new page is started for the second and subsequent entries but never for the first

afterFirst Alias for "notFirst".

If the page layout is wide or landscape and this is set to "true" then it is treated as "notFirst" as a new page is always started when those layouts begin.

For tables which generate multiple files and this is set to either "notfirst" or "afterfirst" then this will not start a new page for the first file but will do for subsequent files. This allows for indexes to be able to start on a single page but then a new page per file.

49.12.6 chapter, section and subsection elements

The **chapter** and **section** elements start the appropriate sectioning before the content. They are optional and only take effect when defined.

The body of the elements contains the title to be used for that section. It can hold expansion expressions, like what Listing 49.10 on page 909 shows.

These elements can work together, so if both **chapter** and **section** are defined together then this will start both a chapter then a section.

toc attribute

Both **chapter** and **section** accept a **toc** attribute which, if defined, will set the text used in the table of contents. If it is not defined or "" then the toc will use the main text in the element's body.

The **toc** attribute is used when there's TEX content in the main title.

e.g. if you have \\in the main text like in Listing 49.10 on the next page, that would break the table of contents so **toc** is used with a version without that formatting.

```
<section toc="OSBYTE 0x$(hex) ($(hex|toDec)) $(name)">
    OSBYTE 0x$(hex) ($(hex|toDec))\\$(name)
</section>
```

Listing 49.10: Example of section element

subsection element

Although unused, the **subsection** element is also available to start a LATEX subsection.

It is identical to the other two elements in how it's used and can interact with them.

However, if **chapter** is defined with it then you should also define **section** otherwise the subsections will appear in the document within a section 0 and you will get errors in the TEX log file.

49.12.7 tableType attribute

This defines the type of table generated.

It takes one of the following values:

table and figure are identical other than the outer TEX environment used

table	A standard TEX table
figure	A standard TEX figure
longtable	A long table which can span multiple pages
long	Alias for "longtable"
tabular	A tabular without the containing table or figure environment.
margintable	Like "table" but is located in the margin.
margin	Alias for "margintable"

The default is "table".

If either "longtable" or "tabular" are selected then the **layout** attribute affects the page layout.

For "table", the **layout** attribute determines if it's a normal or wide table.

49.12.8 vfill attribute

If this is "true" then the LATEX \vfill macro is used both before and after the table to center it vertically with other content.

This only applies to normal tables and is unused by the other types.

49.12.9 align attribute

If this is defined, sets the alignment for longtables. It is unused by the other types.

49.12.10 Table rules

The **topRule**, **midRule** and **bottomRule** attributes allow control of the inclusion of rules in the table. The default is to show them unless overruled by these attributes.

topRule

This is the rule at the top of the table, before the header row.

midRule

This is the rule between the header and the first body row.

bottomRule

This is the rule at the bottom of the table, after the last row in the table.

49.12.11 Volume 0 index

These attributes only affect the Volume 0 figure index and are not used in external content.

preview attribute

If set to "true" will cause the content of the table to appear in Volume 0's figure index. When "false", the default, then only an entry appears, not the content.

index attribute

If present defines the index entry in Volume 0.

49.12.12 prefix and suffix Elements

If present contain TEX code to include before or after the table.

49.12.13 indexOn Element

If this element exists then this will cause the entry to be placed into the document's index. The body contains the field name to get the index entry from.

This element can accept three attributes of its own.

name

Defines the index name to add the entry to. If this is not defined then the main alphabetical index is used.

upperCase

If set to true, force the field value to be in upper case when passed to the index.

lowerCase

If set to true, will force the entry to lower case.

49.12.14 orderBy Element

The **orderBy** element defines how the table's rows are ordered. This element takes five attributes.

descending

If "true" will order the table in descending order rather than the default ascending order.

caseInsensitive

If "true" will cause both upper and lower case letters to be treated the same. If "false" then Upper case will be before Lower case letters.

field

is the only required attribute and defines the field to order against. The optional **then** and **finally** attributes will, if present, define the second and third fields to order by.

49.12.15 field Elements

Each field element defines each field in the table. Most of the configuration are in attributes defined below.

name

Defines the name of the field to use, whilst **title** is the title for the column used in the table's header. If title is not provided then the name in **name** is used.

typewriter

If this is set to "true" then the field will be typeset in the current typewriter/monospaced font.

skipIfBlank

If this is set to "true" it will cause the entire row to be skipped if this field is blank. If multiple columns have this set then if any of them are blank then the row is skipped.

na

If true, will use the **\na** macro for content when it's blank. The default is to show nothing if the field is blank.

align

Defines how the field is aligned when used in a tabular or longtable. It is unused by the other types.

Table 49.3 on page 917 shows the possible options.

rotateHeader and rotateCell

These both take a value in degrees. If the value is non-zero then the respective header or cell will be rotated counter-clockwise by that amount.

rotateHeader is the most common one used, when the column is a single character but we need to describe what that character is. The 6502 instruction lists is an example of this, where we have three columns for the three revisions and a ⋆ showing which revision the instruction relates to. The columns are rotated by 90 degrees.

transform

Will apply a named transform to the field output.

Each transform is a `|` separated list of actions which are listed in Table 49.4 on page 917.

The `|bytes` transform is useful in Machine Code instructions where you want to indicate the position of a number within the instruction.

e.g. for the Z80 the `BIT 0,(ix+d)` instruction has the hex code of "ddcbnn46" where nn represents the offset d.

The `|bytes` transform will force it to uppercase except for the "nn" which is forced to lowercase, so it will appear as "DDCBnn46".

noEscape

will if "true" cause no escaping of the content when translated into TEX. The default is to escape the text ensuring that the output is valid TEX. Use this with caution, but it is needed in a few places.

border

Defines the colour of the cell's border. Not used but still supported.

footNote

Text for a footnote for this cell. Not used but still supported.

49.12.16 dynamic fields

The output of a field can be custom based on the value of a field. This is accomplished with the **match** and **default** child elements.

When present then each match element is checked and if it matches the fields value then it's value is used instead.

If none of them matched the value then, if present, the default value is used. If that is absent then the field's value is used.

An example of this is the 6502 instruction tables where we have three columns, one each for the three major releases of the chip. Each of those columns have a ⋆ if the instruction is supported on that chip revision, or blank if not.

Listing 49.11 on page 917 shows how this was done.

	Name	Action
l	left	Left align
c	center	center horizontally
r	right	Right align
	width	Paragraph *width*

width can be any dimension, e.g. `2cm`

Table 49.3: Available Transforms

Name	Action
toDec	Convert number to decimal
toHex	Convert number to hexadecimal
toUpper	Convert text to upper case
toLower	Convert text to lower case
bytes	Convert to mixed case

Table 49.4: Available Transforms

```
<field name="proc" title="6502" rotateHeader="90
    " noEscape="true">
    <match match="65C02"/>
    <match match="65816"/>
    <default>$\star$</default>
</field>
<field name="proc" title="65C02" rotateHeader="
    90" noEscape="true">
    <match match="65C02">$\star$</match>
    <match match="65816"/>
    <default>$\star$</default>
</field>
<field name="proc" title="65816" rotateHeader="
    90" noEscape="true">
    <default>$\star$</default>
</field>
```

Listing 49.11: Example of match within a field definition

49.13 proc
Root element

The **proc** element is the root element of the xml file. It was originally meant to represent a microprocessor, but now just represents the contained data set. dd

49.13.1 proc attribute

There is only a single attribute called **proc** which is, if defined, used as part of the filename for generating a Hex grid[29].

29: Hex grid Section 49.7 on page 892

49.13.2 Child elements

The **proc** element can contain any of the elements in Table 49.5. All of them can appear multiple times if required.

Table 49.5: List of child elements for generate

Element	Description	Section	
generate	Generate content based on data	49.6	890
mode	Defines an address mode	**??**	**??**
opcode	Defines a row within the dataset	49.8	899
formatType	Defines a field type for format's	49.5	887
format	Defines the bit format of an instruction	49.4	884

Appendices

Page intentionally blank

6502 Instructions

A

Table A.1: 6502 Instructions by Opcode

Opcode	Hex	Length	6502	65C02	65816	Cycles
ADC (dp,X)	61	2	⋆	⋆	⋆	6
ADC sr,S	63	2			⋆	5
ADC dp	65	2	⋆	⋆	⋆	3
ADC [dp]	67	2			⋆	6
ADC #	69	2	⋆	⋆	⋆	2
ADC addr	6D	3	⋆	⋆	⋆	4
ADC (long)	6F	4			⋆	5
ADC (dp),Y	71	2	⋆	⋆	⋆	5
ADC (dp)	72	2		⋆	⋆	5
ADC (sr,S),Y	73	2			⋆	7
ADC dp,X	75	2	⋆	⋆	⋆	4
ADC [dp],Y	77	2			⋆	6
ADC addr,Y	79	3	⋆	⋆	⋆	4
ADC addr,X	7D	3	⋆	⋆	⋆	4
ADC long,X	7F	4			⋆	5
AND (dp,X)	21	2	⋆	⋆	⋆	6
AND sr,S	23	2			⋆	4
AND dp	25	2	⋆	⋆	⋆	3
AND [dp]	27	2			⋆	6
AND #	29	2	⋆	⋆	⋆	2
AND addr	2D	3	⋆	⋆	⋆	4
AND long	2F	4			⋆	5
AND (dp),Y	31	2	⋆	⋆	⋆	5
AND (dp)	32	2		⋆	⋆	5
AND (sr,S),Y	33	2			⋆	7
AND dp,X	35	2	⋆	⋆	⋆	4
AND [dp],Y	37	2			⋆	6
AND addr,Y	39	3	⋆	⋆	⋆	4
AND addr,X	3D	3	⋆	⋆	⋆	4
AND long,X	3F	4			⋆	5
ASL dp	06	2	⋆	⋆	⋆	5
ASL A	0A	1	⋆	⋆	⋆	2

Table A.1: 6502 Instructions by Opcode (continued)

Opcode	Hex	Length	6502	65C02	65816	Cycles
ASL addr	0E	3	⋆	⋆	⋆	6
ASL dp,X	16	2	⋆	⋆	⋆	6
ASL addr,X	1E	3	⋆	⋆	⋆	7
BCC near	90	2	⋆	⋆	⋆	2
BCS near	B0	2	⋆	⋆	⋆	2
BEQ near	F0	2	⋆	⋆	⋆	2
BIT dp	24	2	⋆	⋆	⋆	3
BIT addr	2C	3	⋆	⋆	⋆	4
BIT dp,X	34	2		⋆	⋆	4
BIT addr,X	3C	3		⋆	⋆	4
BIT #	89	2		⋆	⋆	2
BMI near	30	2	⋆	⋆	⋆	2
BNE near	D0	2	⋆	⋆	⋆	2
BPL near	10	2	⋆	⋆	⋆	2
BRA near	80	2		⋆	⋆	3
BRK	00	2	⋆	⋆	⋆	7
BRL near	82	3			⋆	4
BVC near	50	2	⋆	⋆	⋆	2
BVS near	70	2	⋆	⋆	⋆	2
CLC	18	1	⋆	⋆	⋆	2
CLD	D8	1	⋆	⋆	⋆	2
CLI	58	1	⋆	⋆	⋆	2
CLV	B8	1	⋆	⋆	⋆	2
CMP (dp,X)	C1	2	⋆	⋆	⋆	6
CMP sr,S	C3	2			⋆	7
CMP dp	C5	2	⋆	⋆	⋆	3
CMP [dp]	C7	2			⋆	5
CMP #	C9	2	⋆	⋆	⋆	2
CMP addr	CD	3	⋆	⋆	⋆	4
CMP long	CF	4			⋆	5
CMP (dp),Y	D1	2	⋆	⋆	⋆	5
CMP (dp)	D2	2		⋆	⋆	5
CMP (sr,S),Y	D3	2			⋆	7
CMP dp,X	D5	2	⋆	⋆	⋆	4
CMP [dp],Y	D7	2			⋆	6
CMP addr,Y	D9	3	⋆	⋆	⋆	4

Table A.1: 6502 Instructions by Opcode (continued)

Opcode	Hex	Length	6502	65C02	65816	Cycles
CMP addr,X	DD	3	⋆	⋆	⋆	4
CMP long,X	DF	4			⋆	5
COP const	02	2			⋆	7
CPX #	E0	2	⋆	⋆	⋆	2
CPX dp	E4	2	⋆	⋆	⋆	3
CPX addr	EC	3	⋆	⋆	⋆	4
CPY #	C0	2	⋆	⋆	⋆	2
CPY dp	C4	2	⋆	⋆	⋆	3
CPY addr	CC	3	⋆	⋆	⋆	4
DEC A	3A	1	⋆	⋆	⋆	2
DEC dp	C6	2	⋆	⋆	⋆	5
DEC addr	CE	3	⋆	⋆	⋆	6
DEC dp,X	D6	2	⋆	⋆	⋆	6
DEC addr,X	DE	3	⋆	⋆	⋆	7
DEX	CA	1	⋆	⋆	⋆	2
DEY	88	1	⋆	⋆	⋆	2
EOR (dp,X)	41	2	⋆	⋆	⋆	6
EOR sr,S	43	2			⋆	4
EOR dp	45	2	⋆	⋆	⋆	3
EOR [dp]	47	2			⋆	6
EOR #	49	2	⋆	⋆	⋆	2
EOR addr	4D	3	⋆	⋆	⋆	4
EOR long	4F	4			⋆	5
EOR (dp),Y	51	2	⋆	⋆	⋆	5
EOR (dp)	52	2		⋆	⋆	5
EOR (sr,S),Y	53	2			⋆	7
EOR dp,X	55	2	⋆	⋆	⋆	4
EOR [dp],Y	57	2			⋆	6
EOR addr,Y	59	3	⋆	⋆	⋆	4
EOR addr,X	5D	3	⋆	⋆	⋆	4
EOR long,X	5F	4			⋆	5
INC A	1A	1	⋆	⋆	⋆	2
INC dp	E6	2	⋆	⋆	⋆	5
INC addr	EE	3	⋆	⋆	⋆	6
INC dp,X	F6	2	⋆	⋆	⋆	6
INC addr,X	FE	3	⋆	⋆	⋆	7

Table A.1: 6502 Instructions by Opcode (continued)

Opcode	Hex	Length	6502	65C02	65816	Cycles
INX	E8	1	⋆	⋆	⋆	2
INY	C8	1	⋆	⋆	⋆	2
JMP addr	4C	3	⋆	⋆	⋆	4
JMP long	5C	4			⋆	4
JMP (addr)	6C	3	⋆	⋆	⋆	5
JMP (addr,X)	7C	3		⋆	⋆	6
JMP [long]	DC	3			⋆	6
JSR addr	20	3	⋆	⋆	⋆	6
JSR long	22	4			⋆	8
JSR (addr,X)	FC	3			⋆	8
LDA (dp,X)	A1	2	⋆	⋆	⋆	6
LDA sr,S	A3	2			⋆	4
LDA dp	A5	2	⋆	⋆	⋆	3
LDA [dp]	A7	2			⋆	6
LDA #	A9	2	⋆	⋆	⋆	2
LDA addr	AD	3	⋆	⋆	⋆	4
LDA long	AF	3			⋆	4
LDA (dp),Y	B1	2	⋆	⋆	⋆	5
LDA (dp)	B2	2		⋆	⋆	5
LDA (sr,S),Y	B3	2			⋆	7
LDA dp,X	B5	2	⋆	⋆	⋆	4
LDA [dp],Y	B7	2			⋆	6
LDA addr,Y	B9	3	⋆	⋆	⋆	4
LDA addr,X	BD	3	⋆	⋆	⋆	4
LDA long,X	BF	4			⋆	5
LDX #	A2	2	⋆	⋆	⋆	2
LDX dp	A6	2	⋆	⋆	⋆	3
LDX addr	AE	3	⋆	⋆	⋆	4
LDX dp,Y	B6	2	⋆	⋆	⋆	4
LDX addr,Y	BE	3	⋆	⋆	⋆	4
LDY #	A0	2	⋆	⋆	⋆	2
LDY dp	A4	2	⋆	⋆	⋆	3
LDY addr	AC	3	⋆	⋆	⋆	4
LDY dp,X	B4	2	⋆	⋆	⋆	4
LDY addr,X	BC	3	⋆	⋆	⋆	4
LSR dp	46	2	⋆	⋆	⋆	5

Table A.1: 6502 Instructions by Opcode (continued)

Opcode	Hex	Length	6502	65C02	65816	Cycles
LSR A	4A	1	⋆	⋆	⋆	2
LSR addr	4E	3	⋆	⋆	⋆	6
LSR dp,X	56	2	⋆	⋆	⋆	6
LSR addr,X	5E	3	⋆	⋆	⋆	7
MVN src, dest	54	3			⋆	n
MVP src, dest	44	3			⋆	n
NOP	EA	1	⋆	⋆	⋆	2
ORA (dp,X)	01	2	⋆	⋆	⋆	6
ORA sr,S	03	2			⋆	4
ORA dp	05	2	⋆	⋆	⋆	5
ORA [dp]	07	2			⋆	6
ORA #	09	2	⋆	⋆	⋆	2
ORA addr	0D	3	⋆	⋆	⋆	4
ORA long	0F	4			⋆	5
ORA (dp),Y	11	2		⋆	⋆	5
ORA (dp)	12	2			⋆	5
ORA (sr,S),Y	13	2			⋆	7
ORA dp,X	15	2	⋆	⋆	⋆	4
ORA [dp],Y	17	2			⋆	6
ORA addr,Y	19	3	⋆	⋆	⋆	4
ORA addr,X	1D	3	⋆	⋆	⋆	4
ORA long,X	1F	4	⋆	⋆	⋆	5
PEA	F4	3			⋆	5
PEI	D4	2			⋆	6
PER	62	3			⋆	6
PHA	48	1	⋆	⋆	⋆	3
PHB	8B	1			⋆	3
PHD	0B	1			⋆	4
PHK	4B	1			⋆	3
PHP	08	1	⋆	⋆	⋆	4
PHX	DA	1		⋆	⋆	3
PHY	5A	1		⋆	⋆	3
PLA	68	1	⋆	⋆	⋆	3
PLB	AB	1			⋆	4
PLD	2B	1			⋆	5
PLP	28	1	⋆	⋆	⋆	4

Table A.1: 6502 Instructions by Opcode (continued)

Opcode	Hex	Length	6502	65C02	65816	Cycles
PLX	FA	1		⋆	⋆	4
PLY	7A	1		⋆	⋆	4
REP #	C2	2			⋆	3
ROL dp	26	2	⋆	⋆	⋆	5
ROL A	2A	1	⋆	⋆	⋆	2
ROL addr	2E	3	⋆	⋆	⋆	6
ROL dp,X	36	2	⋆	⋆	⋆	6
ROL addr,X	3E	3	⋆	⋆	⋆	7
ROR dp	66	2	⋆	⋆	⋆	5
ROR A	6A	1	⋆	⋆	⋆	2
ROR addr	6E	3	⋆	⋆	⋆	6
ROR dp,X	76	2	⋆	⋆	⋆	6
ROR addr,X	7E	3	⋆	⋆	⋆	7
RTI	40	1	⋆	⋆	⋆	6
RTL	6B	1			⋆	6
RTS	60	1	⋆	⋆	⋆	6
SBC (dp,X)	E1	2	⋆	⋆	⋆	6
SBC sr,S	E3	2			⋆	5
SBC dp	E5	2	⋆	⋆	⋆	3
SBC [dp]	E7	2			⋆	6
SBC #	E9	2	⋆	⋆	⋆	2
SBC addr	ED	3	⋆	⋆	⋆	4
SBC long	EF	4			⋆	5
SBC (dp),Y	F1	2	⋆	⋆	⋆	5
SBC (dp)	F2	2	⋆	⋆	⋆	5
SBC (sr,S),Y	F3	2			⋆	7
SBC dp,X	F5	2	⋆	⋆	⋆	4
SBC [dp],Y	F7	2			⋆	6
SBC addr,Y	F9	3	⋆	⋆	⋆	4
SBC addr,X	FD	3	⋆	⋆	⋆	4
SBC long,X	FF	4			⋆	5
SEC	38	1	⋆	⋆	⋆	2
SED	F8	1	⋆	⋆	⋆	2
SEI	78	1	⋆	⋆	⋆	2
SEP	E2	2			⋆	3
STA (dp,X)	81	2	⋆	⋆	⋆	6

Table A.1: 6502 Instructions by Opcode (continued)

Opcode	Hex	Length	6502	65C02	65816	Cycles
STA sr,S	83	2			⋆	4
STA dp	85	2	⋆	⋆	⋆	3
STA [dp]	87	2			⋆	6
STA addr	8D	3	⋆	⋆	⋆	4
STA long	8F	3			⋆	4
STA (dp),Y	91	2	⋆	⋆	⋆	5
STA (dp)	92	2		⋆	⋆	5
STA (sr,S),Y	93	2			⋆	7
STA dp,X	95	2	⋆	⋆	⋆	4
STA [dp],Y	97	2			⋆	6
STA addr,Y	99	3	⋆	⋆	⋆	4
STA addr,X	9D	3	⋆	⋆	⋆	4
STA long,X	9F	4			⋆	5
STP	DB	1			⋆	2
STX dp	86	2	⋆	⋆	⋆	3
STX addr	8E	3	⋆	⋆	⋆	4
STX dp,Y	96	2	⋆	⋆	⋆	4
STY dp	84	2	⋆	⋆	⋆	3
STY addr	8C	3	⋆	⋆	⋆	4
STY dp,X	94	2	⋆	⋆	⋆	4
STZ dp	64	2		⋆	⋆	3
STZ dp,X	74	2		⋆	⋆	4
STZ addr	9C	3		⋆	⋆	4
STZ addr,X	9E	3		⋆	⋆	5
TAX	AA	1	⋆	⋆	⋆	2
TAY	A8	1	⋆	⋆	⋆	2
TCD	5B	1			⋆	2
TCS	1B	1			⋆	2
TDC	7B	1			⋆	2
TRB dp	14	2		⋆	⋆	5
TRB addr	1C	3		⋆	⋆	6
TSB dp	04	2		⋆	⋆	5
TSB addr	0C	3		⋆	⋆	6
TSC	3B	1			⋆	2
TSX	BA	1	⋆	⋆	⋆	2
TXA	8A	1	⋆	⋆	⋆	2

Table A.1: 6502 Instructions by Opcode (continued)

Opcode	Hex	Length	6502	65C02	65816	Cycles
TXS	9A	1	⋆	⋆	⋆	2
TXY	9B	1			⋆	2
TYA	98	1	⋆	⋆	⋆	2
TYX	BB	1			⋆	2
WAI	CB	1			⋆	3
WDM	42	2			⋆	n
XBA	EB	1			⋆	2
XCE	FB	1			⋆	2

Table A.2: 6502 Instructions by Hex

Hex	Opcode	Length	6502	65C02	65816	Cycles
00	BRK	2	⋆	⋆	⋆	7
01	ORA (dp,X)	2	⋆	⋆	⋆	6
02	COP const	2			⋆	7
03	ORA sr,S	2			⋆	4
04	TSB dp	2		⋆	⋆	5
05	ORA dp	2	⋆	⋆	⋆	5
06	ASL dp	2	⋆	⋆	⋆	5
07	ORA [dp]	2			⋆	6
08	PHP	1	⋆	⋆	⋆	4
09	ORA #	2	⋆	⋆	⋆	2
0A	ASL A	1	⋆	⋆	⋆	2
0B	PHD	1			⋆	4
0C	TSB addr	3		⋆	⋆	6
0D	ORA addr	3	⋆	⋆	⋆	4
0E	ASL addr	3	⋆	⋆	⋆	6
0F	ORA long	4			⋆	5
10	BPL near	2	⋆	⋆	⋆	2
11	ORA (dp),Y	2		⋆	⋆	5
12	ORA (dp)	2			⋆	5
13	ORA (sr,S),Y	2			⋆	7
14	TRB dp	2		⋆	⋆	5
15	ORA dp,X	2	⋆	⋆	⋆	4

Table A.2: 6502 Instructions by Hex (continued)

Hex	Opcode	Length	6502	65C02	65816	Cycles
16	ASL dp,X	2	⋆	⋆	⋆	6
17	ORA [dp],Y	2			⋆	6
18	CLC	1	⋆	⋆	⋆	2
19	ORA addr,Y	3	⋆	⋆	⋆	4
1A	INC A	1	⋆	⋆	⋆	2
1B	TCS	1			⋆	2
1C	TRB addr	3		⋆	⋆	6
1D	ORA addr,X	3	⋆	⋆	⋆	4
1E	ASL addr,X	3	⋆	⋆	⋆	7
1F	ORA long,X	4	⋆	⋆	⋆	5
20	JSR addr	3	⋆	⋆	⋆	6
21	AND (dp,X)	2	⋆	⋆	⋆	6
22	JSR long	4			⋆	8
23	AND sr,S	2			⋆	4
24	BIT dp	2	⋆	⋆	⋆	3
25	AND dp	2	⋆	⋆	⋆	3
26	ROL dp	2	⋆	⋆	⋆	5
27	AND [dp]	2			⋆	6
28	PLP	1	⋆	⋆	⋆	4
29	AND #	2	⋆	⋆	⋆	2
2A	ROL A	1	⋆	⋆	⋆	2
2B	PLD	1			⋆	5
2C	BIT addr	3	⋆	⋆	⋆	4
2D	AND addr	3	⋆	⋆	⋆	4
2E	ROL addr	3	⋆	⋆	⋆	6
2F	AND long	4			⋆	5
30	BMI near	2	⋆	⋆	⋆	2
31	AND (dp),Y	2	⋆	⋆	⋆	5
32	AND (dp)	2		⋆	⋆	5
33	AND (sr,S),Y	2			⋆	7
34	BIT dp,X	2		⋆	⋆	4
35	AND dp,X	2	⋆	⋆	⋆	4
36	ROL dp,X	2	⋆	⋆	⋆	6
37	AND [dp],Y	2			⋆	6
38	SEC	1	⋆	⋆	⋆	2
39	AND addr,Y	3	⋆	⋆	⋆	4

Table A.2: 6502 Instructions by Hex (continued)

Hex	Opcode	Length	6502	65C02	65816	Cycles
3A	DEC A	1	⋆	⋆	⋆	2
3B	TSC	1			⋆	2
3C	BIT addr,X	3		⋆	⋆	4
3D	AND addr,X	3	⋆	⋆	⋆	4
3E	ROL addr,X	3	⋆	⋆	⋆	7
3F	AND long,X	4			⋆	5
40	RTI	1	⋆	⋆	⋆	6
41	EOR (dp,X)	2	⋆	⋆	⋆	6
42	WDM	2			⋆	n
43	EOR sr,S	2			⋆	4
44	MVP src, dest	3			⋆	n
45	EOR dp	2	⋆	⋆	⋆	3
46	LSR dp	2	⋆	⋆	⋆	5
47	EOR [dp]	2			⋆	6
48	PHA	1	⋆	⋆	⋆	3
49	EOR #	2	⋆	⋆	⋆	2
4A	LSR A	1	⋆	⋆	⋆	2
4B	PHK	1			⋆	3
4C	JMP addr	3	⋆	⋆	⋆	4
4D	EOR addr	3	⋆	⋆	⋆	4
4E	LSR addr	3	⋆	⋆	⋆	6
4F	EOR long	4			⋆	5
50	BVC near	2	⋆	⋆	⋆	2
51	EOR (dp),Y	2	⋆	⋆	⋆	5
52	EOR (dp)	2		⋆	⋆	5
53	EOR (sr,S),Y	2			⋆	7
54	MVN src, dest	3			⋆	n
55	EOR dp,X	2	⋆	⋆	⋆	4
56	LSR dp,X	2	⋆	⋆	⋆	6
57	EOR [dp],Y	2			⋆	6
58	CLI	1	⋆	⋆	⋆	2
59	EOR addr,Y	3	⋆	⋆	⋆	4
5A	PHY	1		⋆	⋆	3
5B	TCD	1			⋆	2
5C	JMP long	4			⋆	4
5D	EOR addr,X	3	⋆	⋆	⋆	4

Table A.2: 6502 Instructions by Hex (continued)

Hex	Opcode	Length	6502	65C02	65816	Cycles
5E	LSR addr,X	3	⋆	⋆	⋆	7
5F	EOR long,X	4			⋆	5
60	RTS	1	⋆	⋆	⋆	6
61	ADC (dp,X)	2	⋆	⋆	⋆	6
62	PER	3			⋆	6
63	ADC sr,S	2			⋆	5
64	STZ dp	2		⋆	⋆	3
65	ADC dp	2	⋆	⋆	⋆	3
66	ROR dp	2	⋆	⋆	⋆	5
67	ADC [dp]	2			⋆	6
68	PLA	1	⋆	⋆	⋆	3
69	ADC #	2	⋆	⋆	⋆	2
6A	ROR A	1	⋆	⋆	⋆	2
6B	RTL	1			⋆	6
6C	JMP (addr)	3	⋆	⋆	⋆	5
6D	ADC addr	3	⋆	⋆	⋆	4
6E	ROR addr	3	⋆	⋆	⋆	6
6F	ADC (long)	4			⋆	5
70	BVS near	2	⋆	⋆	⋆	2
71	ADC (dp),Y	2	⋆	⋆	⋆	5
72	ADC (dp)	2		⋆	⋆	5
73	ADC (sr,S),Y	2			⋆	7
74	STZ dp,X	2		⋆	⋆	4
75	ADC dp,X	2	⋆	⋆	⋆	4
76	ROR dp,X	2	⋆	⋆	⋆	6
77	ADC [dp],Y	2			⋆	6
78	SEI	1	⋆	⋆	⋆	2
79	ADC addr,Y	3	⋆	⋆	⋆	4
7A	PLY	1		⋆	⋆	4
7B	TDC	1			⋆	2
7C	JMP (addr,X)	3		⋆	⋆	6
7D	ADC addr,X	3	⋆	⋆	⋆	4
7E	ROR addr,X	3	⋆	⋆	⋆	7
7F	ADC long,X	4			⋆	5
80	BRA near	2		⋆	⋆	3
81	STA (dp,X)	2	⋆	⋆	⋆	6

Table A.2: 6502 Instructions by Hex (continued)

Hex	Opcode	Length	6502	65C02	65816	Cycles
82	BRL near	3			⋆	4
83	STA sr,S	2			⋆	4
84	STY dp	2	⋆	⋆	⋆	3
85	STA dp	2	⋆	⋆	⋆	3
86	STX dp	2	⋆	⋆	⋆	3
87	STA [dp]	2			⋆	6
88	DEY	1	⋆	⋆	⋆	2
89	BIT #	2		⋆	⋆	2
8A	TXA	1	⋆	⋆	⋆	2
8B	PHB	1			⋆	3
8C	STY addr	3	⋆	⋆	⋆	4
8D	STA addr	3	⋆	⋆	⋆	4
8E	STX addr	3	⋆	⋆	⋆	4
8F	STA long	3			⋆	4
90	BCC near	2	⋆	⋆	⋆	2
91	STA (dp),Y	2	⋆	⋆	⋆	5
92	STA (dp)	2		⋆	⋆	5
93	STA (sr,S),Y	2			⋆	7
94	STY dp,X	2	⋆	⋆	⋆	4
95	STA dp,X	2	⋆	⋆	⋆	4
96	STX dp,Y	2	⋆	⋆	⋆	4
97	STA [dp],Y	2			⋆	6
98	TYA	1	⋆	⋆	⋆	2
99	STA addr,Y	3	⋆	⋆	⋆	4
9A	TXS	1	⋆	⋆	⋆	2
9B	TXY	1			⋆	2
9C	STZ addr	3		⋆	⋆	4
9D	STA addr,X	3	⋆	⋆	⋆	4
9E	STZ addr,X	3		⋆	⋆	5
9F	STA long,X	4			⋆	5
A0	LDY #	2	⋆	⋆	⋆	2
A1	LDA (dp,X)	2	⋆	⋆	⋆	6
A2	LDX #	2	⋆	⋆	⋆	2
A3	LDA sr,S	2			⋆	4
A4	LDY dp	2	⋆	⋆	⋆	3
A5	LDA dp	2	⋆	⋆	⋆	3

Table A.2: 6502 Instructions by Hex (continued)

Hex	Opcode	Length	6502	65C02	65816	Cycles
A6	LDX dp	2	⋆	⋆	⋆	3
A7	LDA [dp]	2			⋆	6
A8	TAY	1	⋆	⋆	⋆	2
A9	LDA #	2	⋆	⋆	⋆	2
AA	TAX	1	⋆	⋆	⋆	2
AB	PLB	1			⋆	4
AC	LDY addr	3	⋆	⋆	⋆	4
AD	LDA addr	3	⋆	⋆	⋆	4
AE	LDX addr	3	⋆	⋆	⋆	4
AF	LDA long	3			⋆	4
B0	BCS near	2	⋆	⋆	⋆	2
B1	LDA (dp),Y	2	⋆	⋆	⋆	5
B2	LDA (dp)	2		⋆	⋆	5
B3	LDA (sr,S),Y	2			⋆	7
B4	LDY dp,X	2	⋆	⋆	⋆	4
B5	LDA dp,X	2	⋆	⋆	⋆	4
B6	LDX dp,Y	2	⋆	⋆	⋆	4
B7	LDA [dp],Y	2			⋆	6
B8	CLV	1	⋆	⋆	⋆	2
B9	LDA addr,Y	3	⋆	⋆	⋆	4
BA	TSX	1	⋆	⋆	⋆	2
BB	TYX	1			⋆	2
BC	LDY addr,X	3	⋆	⋆	⋆	4
BD	LDA addr,X	3	⋆	⋆	⋆	4
BE	LDX addr,Y	3	⋆	⋆	⋆	4
BF	LDA long,X	4			⋆	5
C0	CPY #	2	⋆	⋆	⋆	2
C1	CMP (dp,X)	2	⋆	⋆	⋆	6
C2	REP #	2			⋆	3
C3	CMP sr,S	2			⋆	7
C4	CPY dp	2	⋆	⋆	⋆	3
C5	CMP dp	2	⋆	⋆	⋆	3
C6	DEC dp	2	⋆	⋆	⋆	5
C7	CMP [dp]	2			⋆	5
C8	INY	1	⋆	⋆	⋆	2
C9	CMP #	2	⋆	⋆	⋆	2

Table A.2: 6502 Instructions by Hex (continued)

Hex	Opcode	Length	6502	65C02	65816	Cycles
CA	DEX	1	⋆	⋆	⋆	2
CB	WAI	1			⋆	3
CC	CPY addr	3	⋆	⋆	⋆	4
CD	CMP addr	3	⋆	⋆	⋆	4
CE	DEC addr	3	⋆	⋆	⋆	6
CF	CMP long	4			⋆	5
D0	BNE near	2	⋆	⋆	⋆	2
D1	CMP (dp),Y	2	⋆	⋆	⋆	5
D2	CMP (dp)	2		⋆	⋆	5
D3	CMP (sr,S),Y	2			⋆	7
D4	PEI	2			⋆	6
D5	CMP dp,X	2	⋆	⋆	⋆	4
D6	DEC dp,X	2	⋆	⋆	⋆	6
D7	CMP [dp],Y	2			⋆	6
D8	CLD	1	⋆	⋆	⋆	2
D9	CMP addr,Y	3	⋆	⋆	⋆	4
DA	PHX	1		⋆	⋆	3
DB	STP	1			⋆	2
DC	JMP [long]	3			⋆	6
DD	CMP addr,X	3	⋆	⋆	⋆	4
DE	DEC addr,X	3	⋆	⋆	⋆	7
DF	CMP long,X	4			⋆	5
E0	CPX #	2	⋆	⋆	⋆	2
E1	SBC (dp,X)	2	⋆	⋆	⋆	6
E2	SEP	2			⋆	3
E3	SBC sr,S	2			⋆	5
E4	CPX dp	2	⋆	⋆	⋆	3
E5	SBC dp	2	⋆	⋆	⋆	3
E6	INC dp	2	⋆	⋆	⋆	5
E7	SBC [dp]	2			⋆	6
E8	INX	1	⋆	⋆	⋆	2
E9	SBC #	2	⋆	⋆	⋆	2
EA	NOP	1	⋆	⋆	⋆	2
EB	XBA	1			⋆	2
EC	CPX addr	3	⋆	⋆	⋆	4
ED	SBC addr	3	⋆	⋆	⋆	4

Table A.2: 6502 Instructions by Hex (continued)

Hex	Opcode	Length	6502	65C02	65816	Cycles
EE	INC addr	3	⋆	⋆	⋆	6
EF	SBC long	4			⋆	5
F0	BEQ near	2	⋆	⋆	⋆	2
F1	SBC (dp),Y	2	⋆	⋆	⋆	5
F2	SBC (dp)	2	⋆	⋆	⋆	5
F3	SBC (sr,S),Y	2			⋆	7
F4	PEA	3			⋆	5
F5	SBC dp,X	2	⋆	⋆	⋆	4
F6	INC dp,X	2	⋆	⋆	⋆	6
F7	SBC [dp],Y	2			⋆	6
F8	SED	1	⋆	⋆	⋆	2
F9	SBC addr,Y	3	⋆	⋆	⋆	4
FA	PLX	1		⋆	⋆	4
FB	XCE	1			⋆	2
FC	JSR (addr,X)	3			⋆	8
FD	SBC addr,X	3	⋆	⋆	⋆	4
FE	INC addr,X	3	⋆	⋆	⋆	7
FF	SBC long,X	4			⋆	5

A.1 6502 Instruction Grid

hex / len	×0	×1	×2	×3	×4	×5	×6	×7
0×	00 2 **BRK**	01 2 **ORA** (dp,X)	02 2 **COP** const	03 2 **ORA** sr,S	04 2 **TSB** dp	05 2 **ORA** dp	06 2 **ASL** dp	07 2 **ORA** [dp]
1×	10 2 **BPL** near	11 2 **ORA** (dp),Y	12 2 **ORA** (dp)	13 2 **ORA** (sr,S),Y	14 2 **TRB** dp	15 2 **ORA** dp,X	16 2 **ASL** dp,X	17 2 **ORA** [dp],Y
2×	20 3 **JSR** addr	21 2 **AND** (dp,X)	22 4 **JSR** long	23 2 **AND** sr,S	24 2 **BIT** dp	25 2 **AND** dp	26 2 **ROL** dp	27 2 **AND** [dp]
3×	30 2 **BMI** near	31 2 **AND** (dp),Y	32 2 **AND** (dp)	33 2 **AND** (sr,S),Y	34 2 **BIT** dp,X	35 2 **AND** dp,X	36 2 **ROL** dp,X	37 2 **AND** [dp],Y
4×	40 1 **RTI**	41 2 **EOR** (dp,X)	42 2 **WDM**	43 2 **EOR** sr,S	44 3 **MVP** src, dest	45 2 **EOR** dp	46 2 **LSR** dp	47 2 **EOR** [dp]
5×	50 2 **BVC** near	51 2 **EOR** (dp),Y	52 2 **EOR** (dp)	53 2 **EOR** (sr,S),Y	54 3 **MVN** src, dest	55 2 **EOR** dp,X	56 2 **LSR** dp,X	57 2 **EOR** [dp],Y
6×	60 1 **RTS**	61 2 **ADC** (dp,X)	62 3 **PER**	63 2 **ADC** sr,S	64 2 **STZ** dp	65 2 **ADC** dp	66 2 **ROR** dp	67 2 **ADC** [dp]
7×	70 2 **BVS** near	71 2 **ADC** (dp),Y	72 2 **ADC** (dp)	73 2 **ADC** (sr,S),Y	74 2 **STZ** dp,X	75 2 **ADC** dp,X	76 2 **ROR** dp,X	77 2 **ADC** [dp],Y
8×	80 2 **BRA** near	81 2 **STA** (dp,X)	82 3 **BRL** near	83 2 **STA** sr,S	84 2 **STY** dp	85 2 **STA** dp	86 2 **STX** dp	87 2 **STA** [dp]
9×	90 2 **BCC** near	91 2 **STA** (dp),Y	92 2 **STA** (dp)	93 2 **STA** (sr,S),Y	94 2 **STY** dp,X	95 2 **STA** dp,X	96 2 **STX** dp,Y	97 2 **STA** [dp],Y
A×	A0 2 **LDY** #	A1 2 **LDA** (dp,X)	A2 2 **LDX** #	A3 2 **LDA** sr,S	A4 2 **LDY** dp	A5 2 **LDA** dp	A6 2 **LDX** dp	A7 2 **LDA** [dp]
B×	B0 2 **BCS** near	B1 2 **LDA** (dp),Y	B2 2 **LDA** (dp)	B3 2 **LDA** (sr,S),Y	B4 2 **LDY** dp,X	B5 2 **LDA** dp,X	B6 2 **LDX** dp,Y	B7 2 **LDA** [dp],Y
C×	C0 2 **CPY** #	C1 2 **CMP** (dp,X)	C2 2 **REP** #	C3 2 **CMP** sr,S	C4 2 **CPY** dp	C5 2 **CMP** dp	C6 2 **DEC** dp	C7 2 **CMP** [dp]
D×	D0 2 **BNE** near	D1 2 **CMP** (dp),Y	D2 2 **CMP** (dp)	D3 2 **CMP** (sr,S),Y	D4 2 **PEI**	D5 2 **CMP** dp,X	D6 2 **DEC** dp,X	D7 2 **CMP** [dp],Y
E×	E0 2 **CPX** #	E1 2 **SBC** (dp,X)	E2 2 **SEP**	E3 2 **SBC** sr,S	E4 2 **CPX** dp	E5 2 **SBC** dp	E6 2 **INC** dp	E7 2 **SBC** [dp]
F×	F0 2 **BEQ** near	F1 2 **SBC** (dp),Y	F2 2 **SBC** (dp)	F3 2 **SBC** (sr,S),Y	F4 3 **PEA**	F5 2 **SBC** dp,X	F6 2 **INC** dp,X	F7 2 **SBC** [dp],Y

hex len
6502
addr mode

hex len
65C02
addr mode

hex len
65816
addr mode

×8	×9	×A	×B	×C	×D	×E	×F	
08 1 **PHP**	09 2 **ORA** #	0A 1 **ASL** A	0B 1 **PHD**	0C 3 **TSB** addr	0D 3 **ORA** addr	0E 3 **ASL** addr	0F 4 **ORA** long	0×
18 1 **CLC**	19 3 **ORA** addr,Y	1A 1 **INC** A	1B 1 **TCS**	1C 3 **TRB** addr	1D 3 **ORA** addr,X	1E 3 **ASL** addr,X	1F 4 **ORA** long,X	1×
28 1 **PLP**	29 2 **AND** #	2A 1 **ROL** A	2B 1 **PLD**	2C 3 **BIT** addr	2D 3 **AND** addr	2E 3 **ROL** addr	2F 4 **AND** long	2×
38 1 **SEC**	39 3 **AND** addr,Y	3A 1 **DEC** A	3B 1 **TSC**	3C 3 **BIT** addr,X	3D 3 **AND** addr,X	3E 3 **ROL** addr,X	3F 4 **AND** long,X	3×
48 1 **PHA**	49 2 **EOR** #	4A 1 **LSR** A	4B 1 **PHK**	4C 3 **JMP** addr	4D 3 **EOR** addr	4E 3 **LSR** addr	4F 4 **EOR** long	4×
58 1 **CLI**	59 3 **EOR** addr,Y	5A 1 **PHY**	5B 1 **TCD**	5C 4 **JMP** long	5D 3 **EOR** addr,X	5E 3 **LSR** addr,X	5F 4 **EOR** long,X	5×
68 1 **PLA**	69 2 **ADC** #	6A 1 **ROR** A	6B 1 **RTL**	6C 3 **JMP** (addr)	6D 3 **ADC** addr	6E 3 **ROR** addr	6F 4 **ADC** (long)	6×
78 1 **SEI**	79 3 **ADC** addr,Y	7A 1 **PLY**	7B 1 **TDC**	7C 3 **JMP** (addr,X)	7D 3 **ADC** addr,X	7E 3 **ROR** addr,X	7F 4 **ADC** long,X	7×
88 1 **DEY**	89 2 **BIT** #	8A 1 **TXA**	8B 1 **PHB**	8C 3 **STY** addr	8D 3 **STA** addr	8E 3 **STX** addr	8F 3 **STA** long	8×
98 1 **TYA**	99 3 **STA** addr,Y	9A 1 **TXS**	9B 1 **TXY**	9C 3 **STZ** addr	9D 3 **STA** addr,X	9E 3 **STZ** addr,X	9F 4 **STA** long,X	9×
A8 1 **TAY**	A9 2 **LDA** #	AA 1 **TAX**	AB 1 **PLB**	AC 3 **LDY** addr	AD 3 **LDA** addr	AE 3 **LDX** addr	AF 3 **LDA** long	A×
B8 1 **CLV**	B9 3 **LDA** addr,Y	BA 1 **TSX**	BB 1 **TYX**	BC 3 **LDY** addr,X	BD 3 **LDA** addr,X	BE 3 **LDX** addr,Y	BF 4 **LDA** long,X	B×
C8 1 **INY**	C9 2 **CMP** #	CA 1 **DEX**	CB 1 **WAI**	CC 3 **CPY** addr	CD 3 **CMP** addr	CE 3 **DEC** addr	CF 4 **CMP** long	C×
D8 1 **CLD**	D9 3 **CMP** addr,Y	DA 1 **PHX**	DB 1 **STP**	DC 3 **JMP** [long]	DD 3 **CMP** addr,X	DE 3 **DEC** addr,X	DF 4 **CMP** long,X	D×
E8 1 **INX**	E9 2 **SBC** #	EA 1 **NOP**	EB 1 **XBA**	EC 3 **CPX** addr	ED 3 **SBC** addr	EE 3 **INC** addr	EF 4 **SBC** long	E×
F8 1 **SED**	F9 3 **SBC** addr,Y	FA 1 **PLX**	FB 1 **XCE**	FC 3 **JSR** (addr,X)	FD 3 **SBC** addr,X	FE 3 **INC** addr,X	FF 4 **SBC** long,X	F×

hex	len	
6502		addr mode
hex	len	
65C02		addr mode
hex	len	
65816		addr mode

Page intentionally blank

Z80 Instructions B

B.1 Z80 Instruction Grid

The following tables show the Z80 instructions laid out in a 16×16 grid. As some instructions are defined by a prefix, those are shown in a separate table just for that prefix.

Table B.1: Z80 Instruction Grids

Instruction	
`0x`*nn*	Appendix B.1.1 on page 942
`0xCB`*nn*	Appendix B.1.2 on page 944
`0xDD`*nn*	Appendix B.1.3 on page 946
`0xED`*nn*	Appendix B.1.4 on page 948
`0xFD`*nn*	Appendix B.1.5 on page 950

The `0xDD` and `0xFD` prefixes

The `0xDD` and `0xFD` prefixes differenciate the instructions that reference HL with those that reference either IX or IY.

B.1.1 Instructions with no prefix

hex Op params / hex Prefix	×0	×1	×2	×3	×4	×5	×6	×7
0×	00 1 **NOP**	01 3 **LD** bc,NN	02 1 **LD** (bc),a	03 1 **INC** bc	04 1 **INC** b	05 1 **DEC** b	06 1 **LD** b,n	07 1 **RLCA**
1×	10 2 **DJNZ** DIS	11 3 **LD** de,NN	12 1 **LD** (de),a	13 1 **INC** de	14 1 **INC** d	15 1 **DEC** d	16 1 **LD** d,n	17 1 **RLA**
2×	20 2 **JR** nz,DIS	21 3 **LD** hl,NN	22 3 **LD** (NN),hl	23 1 **INC** hl	24 1 **INC** h	25 1 **DEC** h	26 1 **LD** h,n	27 1 **DAA**
3×	30 2 **JR** nc,DIS	31 3 **LD** sp,NN	32 3 **LD** (NN),a	33 1 **INC** sp	34 1 **INC** (hl)	35 1 **DEC** (hl)	36 2 **LD** (hl),n	37 1 **SCF**
4×	40 1 **LD** b,b	41 1 **LD** b,c	42 1 **LD** b,d	43 1 **LD** b,e	44 1 **LD** b,h	45 1 **LD** b,l	46 1 **LD** b,(hl)	47 1 **LD** b,a
5×	50 1 **LD** d,b	51 1 **LD** d,c	52 1 **LD** d,d	53 1 **LD** d,e	54 1 **LD** d,h	55 1 **LD** d,l	56 1 **LD** d,(hl)	57 1 **LD** d,a
6×	60 1 **LD** h,b	61 1 **LD** h,c	62 1 **LD** h,d	63 1 **LD** h,e	64 1 **LD** h,h	65 1 **LD** h,l	66 1 **LD** h,(hl)	67 1 **LD** h,a
7×	70 1 **LD** (hl),b	71 1 **LD** (hl),c	72 1 **LD** (hl),d	73 1 **LD** (hl),e	74 1 **LD** (hl),h	75 1 **LD** (hl),l	76 1 **HALT**	77 1 **LD** (hl),a
8×	80 1 **ADD** a,b	81 1 **ADD** a,c	82 1 **ADD** a,d	83 1 **ADD** a,e	84 1 **ADD** a,h	85 1 **ADD** a,l	86 1 **ADD** a,(hl)	87 1 **ADD** a,a
9×	90 1 **SUB** a,b	91 1 **SUB** a,c	92 1 **SUB** a,d	93 1 **SUB** a,e	94 1 **SUB** a,h	95 1 **SUB** a,l	96 1 **SUB** a,(hl)	97 1 **SUB** a,a
A×	A0 1 **AND** b	A1 1 **AND** c	A2 1 **AND** d	A3 1 **AND** e	A4 1 **AND** h	A5 1 **AND** l	A6 1 **AND** (hl)	A7 1 **AND** a
B×	B0 1 **OR** b	B1 1 **OR** c	B2 1 **OR** d	B3 1 **OR** e	B4 1 **OR** h	B5 1 **OR** l	B6 1 **OR** (hl)	B7 1 **OR** a
C×	C0 1 **RET** nz	C1 1 **POP** bc	C2 3 **JP** nz,NN	C3 3 **JP** NN	C4 3 **CALL** nz,NN	C5 1 **PUSH** bc	C6 2 **ADD** a,N	C7 1 **RST** 0
D×	D0 1 **RET** nc	D1 1 **POP** de	D2 3 **JP** nc,NN	D3 1 **OUT** (N),a	D4 3 **CALL** nc,NN	D5 1 **PUSH** de	D6 2 **SUB** N	D7 1 **RST** 16
E×	E0 1 **RET** po	E1 1 **POP** hl	E2 3 **JP** po,NN	E3 1 **EX** (sp),hl	E4 3 **CALL** po,NN	E5 1 **PUSH** hl	E6 2 **AND** N	E7 1 **RST** 32
F×	F0 1 **RET** p	F1 1 **POP** af	F2 3 **JP** p,NN	F3 1 **DI**	F4 3 **CALL** p,NN	F5 1 **PUSH** af	F6 2 **OR** N	F7 1 **RST** 48

×8	×9	×A	×B	×C	×D	×E	×F	
08 1 **EX** af,af'	09 1 **ADD** hl,bc	0A 1 **LD** a,(bc)	0B 1 **DEC** bc	0C 1 **INC** c	0D 1 **DEC** c	0E 1 **LD** c,N	0F 1 **RRCA**	0×
18 2 **JR** DIS	19 1 **ADD** hl,de	1A 1 **LD** a,(de)	1B 1 **DEC** de	1C 1 **INC** e	1D 1 **DEC** e	1E 1 **LD** e,N	1F 1 **RRA**	1×
28 2 **JR** z,DIS	29 1 **ADD** hl,hl	2A 3 **LD** hl,(NN)	2B 1 **DEC** hl	2C 1 **INC** l	2D 1 **DEC** l	2E 1 **LD** l,N	2F 1 **CPL**	2×
38 2 **JR** c,DIS	39 1 **ADD** hl,sp	3A 3 **LD** a,(NN)	3B 1 **DEC** sp	3C 1 **INC** a	3D 1 **DEC** a	3E 1 **LD** a,N	3F 1 **CCF**	3×
48 1 **LD** c,b	49 1 **LD** c,c	4A 1 **LD** c,d	4B 1 **LD** c,e	4C 1 **LD** c,h	4D 1 **LD** c,l	4E 1 **LD** c,(hl)	4F 1 **LD** c,a	4×
58 1 **LD** e,b	59 1 **LD** e,c	5A 1 **LD** e,d	5B 1 **LD** e,e	5C 1 **LD** e,h	5D 1 **LD** e,l	5E 1 **LD** e,(hl)	5F 1 **LD** e,a	5×
68 1 **LD** l,b	69 1 **LD** l,c	6A 1 **LD** l,d	6B 1 **LD** l,e	6C 1 **LD** l,h	6D 1 **LD** l,l	6E 1 **LD** l,(hl)	6F 1 **LD** l,a	6×
78 1 **LD** a,b	79 1 **LD** a,c	7A 1 **LD** a,d	7B 1 **LD** a,e	7C 1 **LD** a,h	7D 1 **LD** a,l	7E 1 **LD** a,(hl)	7F 1 **LD** a,a	7×
88 1 **ADC** a,b	89 1 **ADC** a,c	8A 1 **ADC** a,d	8B 1 **ADC** a,e	8C 1 **ADC** a,h	8D 1 **ADC** a,l	8E 1 **ADC** a,(hl)	8F 1 **ADC** a,a	8×
98 1 **SBC** a,b	99 1 **SBC** a,c	9A 1 **SBC** a,d	9B 1 **SBC** a,e	9C 1 **SBC** a,h	9D 1 **SBC** a,l	9E 1 **SBC** a,(hl)	9F 1 **SBC** a,a	9×
A8 1 **XOR** b	A9 1 **XOR** c	AA 1 **XOR** d	AB 1 **XOR** e	AC 1 **XOR** h	AD 1 **XOR** l	AE 1 **XOR** (hl)	AF 1 **XOR** a	A×
B8 1 **CP** b	B9 1 **CP** c	BA 1 **CP** d	BB 1 **CP** e	BC 1 **CP** h	BD 1 **CP** l	BE 1 **CP** (hl)	BF 1 **CP** a	B×
C8 1 **RET** z	C9 1 **RET**	CA 3 **JP** z,NN	CB **CBNN**	CC 3 **CALL** z,NN	CD 3 **CALL** NN	CE 2 **ADC** a,N	CF 1 **RST** 8	C×
D8 1 **RET** c	D9 1 **EXX**	DA 3 **JP** c,NN	DB 2 **IN** a,(N)	DC 3 **CALL** c,NN	DD **DDNN**	DE 2 **SBC** a,N	DF 1 **RST** 24	D×
E8 1 **RET** pe	E9 1 **JP** (hl)	EA 3 **JP** pe,NN	EB 1 **EX** de,hl	EC 3 **CALL** pe,NN	ED **EDNN**	EE 2 **XOR** N	EF 1 **RST** 40	E×
F8 1 **RET** m	F9 1 **LD** sp,hl	FA 3 **JP** m,NN	FB 1 **EI**	FC 3 **CALL** m,NN	FD **FDNN**	FE 2 **CP** N	FF 1 **RST** 56	F×

hex len
Op
params

hex
Prefix

B.1.2 Instructions after `0xCB`

	×0	×1	×2	×3	×4	×5	×6	×7
0×	CB00 2 **RLC** b	CB01 2 **RLC** c	CB02 2 **RLC** d	CB03 2 **RLC** e	CB04 2 **RLC** h	CB05 2 **RLC** l	CB06 2 **RLC** (hl)	CB07 2 **RLC** a
1×	CB10 2 **RL** b	CB11 2 **RL** c	CB12 2 **RL** d	CB13 2 **RL** e	CB14 2 **RL** h	CB15 2 **RL** l	CB16 2 **RL** (hl)	CB17 2 **RL** a
2×	CB20 2 **SLA** b	CB21 2 **SLA** c	CB22 2 **SLA** d	CB23 2 **SLA** e	CB24 2 **SLA** h	CB25 2 **SLA** l	CB26 2 **SLA** (hl)	CB27 2 **SLA** a
3×								
4×	CB40 2 **BIT** 0,b	CB41 2 **BIT** 0,c	CB42 2 **BIT** 0,d	CB43 2 **BIT** 0,e	CB44 2 **BIT** 0,h	CB45 2 **BIT** 0,l	CB46 2 **BIT** 0,(hl)	CB47 2 **BIT** 0,a
5×	CB50 2 **BIT** 2,b	CB51 2 **BIT** 2,c	CB52 2 **BIT** 2,d	CB53 2 **BIT** 2,e	CB54 2 **BIT** 2,h	CB55 2 **BIT** 2,l	CB56 2 **BIT** 2,(hl)	CB57 2 **BIT** 2,a
6×	CB60 2 **BIT** 4,b	CB61 2 **BIT** 4,c	CB62 2 **BIT** 4,d	CB63 2 **BIT** 4,e	CB64 2 **BIT** 4,h	CB65 2 **BIT** 4,l	CB66 2 **BIT** 4,(hl)	CB67 2 **BIT** 4,a
7×	CB70 2 **BIT** 6,b	CB71 2 **BIT** 6,c	CB72 2 **BIT** 6,d	CB73 2 **BIT** 6,e	CB74 2 **BIT** 6,h	CB75 2 **BIT** 6,l	CB76 2 **BIT** 6,(hl)	CB77 2 **BIT** 6,a
8×	CB80 2 **RES** 0,b	CB81 2 **RES** 0,c	CB82 2 **RES** 0,d	CB83 2 **RES** 0,e	CB84 2 **RES** 0,h	CB85 2 **RES** 0,l	CB86 2 **RES** 0,(hl)	CB87 2 **RES** 0,a
9×	CB90 2 **RES** 2,b	CB91 2 **RES** 2,c	CB92 2 **RES** 2,d	CB93 2 **RES** 2,e	CB94 2 **RES** 2,h	CB95 2 **RES** 2,l	CB96 2 **RES** 2,(hl)	CB97 2 **RES** 2,a
A×	CBA0 2 **RES** 4,b	CBA1 2 **RES** 4,c	CBA2 2 **RES** 4,d	CBA3 2 **RES** 4,e	CBA4 2 **RES** 4,h	CBA5 2 **RES** 4,l	CBA6 2 **RES** 4,(hl)	CBA7 2 **RES** 4,a
B×	CBB0 2 **RES** 6,b	CBB1 2 **RES** 6,c	CBB2 2 **RES** 6,d	CBB3 2 **RES** 6,e	CBB4 2 **RES** 6,h	CBB5 2 **RES** 6,l	CBB6 2 **RES** 6,(hl)	CBB7 2 **RES** 6,a
C×	CBC0 2 **SET** 0,b	CBC1 2 **SET** 0,c	CBC2 2 **SET** 0,d	CBC3 2 **SET** 0,e	CBC4 2 **SET** 0,h	CBC5 2 **SET** 0,l	CBC6 2 **SET** 0,(hl)	CBC7 2 **SET** 0,a
D×	CBD0 2 **SET** 2,b	CBD1 2 **SET** 2,c	CBD2 2 **SET** 2,d	CBD3 2 **SET** 2,e	CBD4 2 **SET** 2,h	CBD5 2 **SET** 2,l	CBD6 2 **SET** 2,(hl)	CBD7 2 **SET** 2,a
E×	CBE0 2 **SET** 4,b	CBE1 2 **SET** 4,c	CBE2 2 **SET** 4,d	CBE3 2 **SET** 4,e	CBE4 2 **SET** 4,h	CBE5 2 **SET** 4,l	CBE6 2 **SET** 4,(hl)	CBE7 2 **SET** 4,a
F×	CBF0 2 **SET** 6,b	CBF1 2 **SET** 6,c	CBF2 2 **SET** 6,d	CBF3 2 **SET** 6,e	CBF4 2 **SET** 6,h	CBF5 2 **SET** 6,l	CBF6 2 **SET** 6,(hl)	CBF7 2 **SET** 6,a

×8	×9	×A	×B	×C	×D	×E	×F	
CB08 2 **RRC** b	CB09 2 **RRC** c	CB0A 2 **RRC** d	CB0B 2 **RRC** e	CB0C 2 **RRC** h	CB0D 2 **RRC** l	CB0E 2 **RRC** (hl)	CB0F 2 **RRC** a	0×
CB18 2 **RR** b	CB19 2 **RR** c	CB1A 2 **RR** d	CB1B 2 **RR** e	CB1C 2 **RR** h	CB1D 2 **RR** l	CB1E 2 **RR** (hl)	CB1F 2 **RR** a	1×
CB28 2 **SRA** b	CB29 2 **SRA** c	CB2A 2 **SRA** d	CB2B 2 **SRA** e	CB2C 2 **SRA** h	CB2D 2 **SRA** l	CB2E 2 **SRA** (hl)	CB2F 2 **SRA** a	2×
CB38 2 **SRL** b	CB39 2 **SRL** c	CB3A 2 **SRL** d	CB3B 2 **SRL** e	CB3C 2 **SRL** h	CB3D 2 **SRL** l	CB3E 2 **SRL** (hl)	CB3F 2 **SRL** a	3×
CB48 2 **BIT** 1,b	CB49 2 **BIT** 1,c	CB4A 2 **BIT** 1,d	CB4B 2 **BIT** 1,e	CB4C 2 **BIT** 1,h	CB4D 2 **BIT** 1,l	CB4E 2 **BIT** 1,(hl)	CB4F 2 **BIT** 1,a	4×
CB58 2 **BIT** 3,b	CB59 2 **BIT** 3,c	CB5A 2 **BIT** 3,d	CB5B 2 **BIT** 3,e	CB5C 2 **BIT** 3,h	CB5D 2 **BIT** 3,l	CB5E 2 **BIT** 3,(hl)	CB5F 2 **BIT** 3,a	5×
CB68 2 **BIT** 5,b	CB69 2 **BIT** 5,c	CB6A 2 **BIT** 5,d	CB6B 2 **BIT** 5,e	CB6C 2 **BIT** 5,h	CB6D 2 **BIT** 5,l	CB6E 2 **BIT** 5,(hl)	CB6F 2 **BIT** 5,a	6×
CB78 2 **BIT** 7,b	CB79 2 **BIT** 7,c	CB7A 2 **BIT** 7,d	CB7B 2 **BIT** 7,e	CB7C 2 **BIT** 7,h	CB7D 2 **BIT** 7,l	CB7E 2 **BIT** 7,(hl)	CB7F 2 **BIT** 7,a	7×
CB88 2 **RES** 1,b	CB89 2 **RES** 1,c	CB8A 2 **RES** 1,d	CB8B 2 **RES** 1,e	CB8C 2 **RES** 1,h	CB8D 2 **RES** 1,l	CB8E 2 **RES** 1,(hl)	CB8F 2 **RES** 1,a	8×
CB98 2 **RES** 3,b	CB99 2 **RES** 3,c	CB9A 2 **RES** 3,d	CB9B 2 **RES** 3,e	CB9C 2 **RES** 3,h	CB9D 2 **RES** 3,l	CB9E 2 **RES** 3,(hl)	CB9F 2 **RES** 3,a	9×
CBA8 2 **RES** 5,b	CBA9 2 **RES** 5,c	CBAA 2 **RES** 5,d	CBAB 2 **RES** 5,e	CBAC 2 **RES** 5,h	CBAD 2 **RES** 5,l	CBAE 2 **RES** 5,(hl)	CBAF 2 **RES** 5,a	A×
CBB8 2 **RES** 7,b	CBB9 2 **RES** 7,c	CBBA 2 **RES** 7,d	CBBB 2 **RES** 7,e	CBBC 2 **RES** 7,h	CBBD 2 **RES** 7,l	CBBE 2 **RES** 7,(hl)	CBBF 2 **RES** 7,a	B×
CBC8 2 **SET** 1,b	CBC9 2 **SET** 1,c	CBCA 2 **SET** 1,d	CBCB 2 **SET** 1,e	CBCC 2 **SET** 1,h	CBCD 2 **SET** 1,l	CBCE 2 **SET** 1,(hl)	CBCF 2 **SET** 1,a	C×
CBD8 2 **SET** 3,b	CBD9 2 **SET** 3,c	CBDA 2 **SET** 3,d	CBDB 2 **SET** 3,e	CBDC 2 **SET** 3,h	CBDD 2 **SET** 3,l	CBDE 2 **SET** 3,(hl)	CBDF 2 **SET** 3,a	D×
CBE8 2 **SET** 5,b	CBE9 2 **SET** 5,c	CBEA 2 **SET** 5,d	CBEB 2 **SET** 5,e	CBEC 2 **SET** 5,h	CBED 2 **SET** 5,l	CBEE 2 **SET** 5,(hl)	CBEF 2 **SET** 5,a	E×
CBF8 2 **SET** 7,b	CBF9 2 **SET** 7,c	CBFA 2 **SET** 7,d	CBFB 2 **SET** 7,e	CBFC 2 **SET** 7,h	CBFD 2 **SET** 7,l	CBFE 2 **SET** 7,(hl)	CBFF 2 **SET** 7,a	F×

B.1.3 Instructions after `0xDD`

	×0	×1	×2	×3	×4	×5	×6	×7
0×								
1×								
2×				DD23 2 **INC** ix				
3×								
4×								
5×								
6×								
7×								
8×								
9×								
A×								
B×								
C×								
D×								
E×		DDE1 2 **POP** ix		DDE3 2 **EX** (sp),ix		DDE5 2 **PUSH** ix		
F×								

×8	×9	×A	×B	×C	×D	×E	×F	
	DD09 2 **ADD** ix,bc							0×
	DD19 2 **ADD** ix,de							1×
	DD29 2 **ADD** ix,ix		DD2B 2 **DEC** ix					2×
	DD39 2 **ADD** ix,sp							3×
								4×
								5×
								6×
								7×
								8×
								9×
								A×
								B×
								C×
								D×
	DDE9 2 **JP** (ix)							E×
	DDF9 2 **LD** sp,ix							F×

B.1.4 Instructions after `0xED`

	×0	×1	×2	×3	×4	×5	×6	×7
0×								
1×								
2×								
3×								
4×	ED40 2 **IN** b,(c)	ED41 2 **OUT** (c),b	ED42 2 **SBC** hl,bc		ED44 2 **NEG**	ED45 2 **RETN**	ED46 2 **IM** 0	ED47 2 **LD** i,a
5×	ED50 2 **IN** d,(c)	ED51 2 **OUT** (c),d	ED52 2 **SBC** hl,de				ED56 2 **IM** 1	ED57 2 **LD** a,i
6×	ED60 2 **IN** h,(c)	ED61 2 **OUT** (c),h	ED62 2 **SBC** hl,hl					ED67 **RRD**
7×	ED70 2 **IN** f,(c)		ED72 2 **SBC** hl,sp					
8×								
9×								
A×	EDA0 2 **LDI**	EDA1 2 **CPI**	EDA2 2 **INI**	EDA3 2 **OUTI**				
B×	EDB0 2 **LDIR**	EDB1 2 **CPIR**	EDB2 2 **INIR**	EDB3 2 **OTIR**				
C×								
D×								
E×								
F×								

×8	×9	×A	×B	×C	×D	×E	×F	
								0×
								1×
								2×
								3×
ED48 2 **IN** c,(c)	ED49 2 **OUT** (c),c	ED4A 2 **ADC** hl,bc	ED4B 2 **LD** bc,(NN)		ED4D 2 **RETI**		ED4F 2 **LD** r,a	4×
ED58 2 **IN** e,(c)	ED59 2 **OUT** (c),e	ED5A 2 **ADC** hl,de				ED5E 2 **IM** 2	ED5F 2 **LD** a,r	5×
ED68 2 **IN** l,(c)	ED69 2 **OUT** (c),l	ED6A 2 **ADC** hl,hl					ED6F 2 **RLD**	6×
ED78 2 **IN** a,(c)	ED79 2 **OUT** (c),a	ED7A 2 **ADC** hl,sp						7×
								8×
								9×
EDA8 2 **LDD**	EDA9 2 **CPD**	EDAA 2 **IND**	EDAB 2 **OUTD**					A×
EDB8 2 **LDDR**	EDB9 2 **CPDR**	EDBA 2 **INDR**	EDBB 2 **OTDR**					B×
								C×
								D×
								E×
								F×

B.1.5 Instructions after `0xFD`

	×0	×1	×2	×3	×4	×5	×6	×7
0×								
1×								
2×				FD23 2 **INC** iy				
3×								
4×								
5×								
6×								
7×								
8×								
9×								
A×								
B×								
C×								
D×								
E×		FDE1 2 **POP** iy		FDE3 2 **EX** (sp),iy		FDE5 2 **PUSH** iy		
F×								

×8	×9	×A	×B	×C	×D	×E	×F	
	FD09 2 **ADD** iy,bc							0×
	FD19 2 **ADD** iy,de							1×
	FD29 2 **ADD** iy,iy		FD2B 2 **DEC** iy					2×
	FD39 2 **ADD** iy,sp							3×
								4×
								5×
								6×
								7×
								8×
								9×
								A×
								B×
								C×
								D×
	FDE9 2 **JP** (iy)							E×
	FDF9 2 **LD** sp,iy							F×

Page intentionally blank

Z80 Instruction Decoding Tables

C

The tables within this Appendix allow you to decode the Z80 instruction set based on the Hex have at the address you are looking at. Each instruction, is organized by bits with one to 4 bytes shown for each instruction.

C.1 How to use the decoding tables

To use the tables, first look at the first byte of the instruction you want to decode. If it's one of the prefixes in Table C.1 on page 956 then the byte is a prefix to an extended instruction. In that instance you should look at the relevant table for the instruction.

If it is not then use the first table,Appendix C.7 on page 959

For each table simply lookup the first byte[1].

The tables are ordered so that exact binary comes first. When an option appears then that gets sorted next, so you should locate the first complete match first, then lookup for the option[2].

1: The tables for the prefixes exclude the prefix so lookup the second byte

2: Not all options use every combination.

e.g. **r** does not use `110`.

C.1.1 Examples

```
0x06 0x42
```

As this does not use a prefix then we use the default table[3]. We scan down from the start, and we find that no exact match for `0x06` `00000110` exists. When we reach the first entries with options, we find that the entry for **LD r,n** matches with the prefix `00`, then r and finally `110`. As we have r as `000` which decodes to the B register, and byte 1 is n the final instruction is:

3: default table Appendix C.7 on page 959

```
06 42       LD B,0x42
```

0xED 0x78

Here as the first byte is the prefix `0xED` we use the ED table[4] and search for the second byte `0x78`. Scanning that table shows that the instruction **IN r,(C)** matches with r being `111` which decodes to the accumulator so the final instruction is:

```
ED 78       IN A,(C)
```

4: ED table Appendix C.10 on page 962

0xDD 0xCB 0x42 0x56

Here as the first byte is the prefix `0xDD` we use the DD table[5] and search for the second byte `0xCB`. Although the second byte itself a prefix, we still use the DD table.

5: DD table Appendix C.9 on page 961

Scanning that table shows that the instruction **BIT b,(IX+d)** matches with byte 1 being `0xCB` and byte 3 matching `0x56`. Byte 2 is the offset d, in this case `0x42`.

This might seem weird that part of the decoding is done in reverse but this is the case for instructions that have an offset from either IX or IY. In either case, the third byte (+2) is the offset from the register's value and the fourth byte (+3) the actual instruction.

So for this, the last byte maps to b `010` so the decoded instruction is: r being `111` which decodes to the accumulator so the final instruction is:

```
DD CB 42 56 BIT 2,(IX + 0x42)
```

C.2 What use are these tables?

You might wonder why have these and also have every instruction listed by its hexcode which is quicker to use.

The reason is to get an idea on why instructions are numbered as they are. To a beginner, the instructions appear to be placed all over the place, but with these tables, you start to see a pattern on how the original chip designers originally created the instruction sets.

After a while you will see patterns within patterns here. For example with the arithmetic instructions, you will notice that **ADD**, **ADC** differ by just 1 bit, bit-3, indicating the use of the Carry flag but also that **SUB** and **SBC** differ not only by the same bit but also bit-4 defines the difference between both sets, with 0 for addition and 1 for subtraction. Hence, you could reduce the basic arithmetic instructions down to `100x crrr` where x=0 for add, x=1 for subtract, c=1 if carry is used and r the register used.

You will notice similarities between other similar instructions like the logic **AND**, **OR** and **XOR** with the pattern `101xxrrr`. Here xx is `00` for **AND**, `01` for **OR** and `10` for **XOR**.

xx=11 is used by **CP**

So, if you wanted to write a cycle accurate Z80 emulator, more so in hardware[6], then you would want to know about these relationships rather than using a lookup table.

6: These exists, including some hardware chip replacement projects or MiSTer

C.3 Instruction Decoding tables

The decode tables are split into five sections with the first section representing instructions with no prefix and the remainder for each prefix. These prefixes are listed in Table C.1 on page 956.

Each table shows up to three bytes per instruction. The first table shows bytes 0, 1 and 2 as there is no prefix, however the remaining tables only show bytes 1, 2 and 3 as the first byte is always the prefix.

Table C.1: Z80 Instruction prefixes

Prefix	Instructions	Table	Page
None	Standard instructions	C.7	959
`0xCB`	Additional instructions	C.8	960
`0xDD`	Instructions using IX	C.9	961
`0xED`	Additional instructions	C.10	962
`0xFD`	Instructions using IY	C.11	963

Prefixes DD and FD can be followed with CB.
In those cases use the relevant DD or FD table for the instruction.

C.3.1 Bit types

The following tables show the possible values of and the meaning of the various bit's defined in the decoding tables. The majority of these use every possible combination.

The exception to this is the `r` and `r'` where the combination `110` is not used by that instruction.

Table C.2: Generic bit types

Bit type	Content	Description
`n`	Immediate Data	Constant value
`a`	Address	16-bit Address
`e`	Relative Address	Distance from PC
`d`	Data Offset	Offset from IX or IY

These entries usually occupy all 8-bits of the byte. `a` takes up 2 bytes, low byte first then high.

Table C.3: Register types **r**, **r'** and **s**

`r` or `r'`	Register	`s`	Register Pair
`000`	B	`00`	BC
`001`	C	`01`	DE
`010`	D	`10`	HL
`011`	E	`11`	SP
`100`	H		
`101`	L		
`111`	A		

r = `110` is invalid and usually not part of the instruction being decoded.

Table C.4: Condition type **c** and **C**

C	c	Condition	
00	000	NZ	Not Zero
01	001	Z	Zero
10	010	NC	Not Carry
11	011	C	Carry
	100	PO	Parity Odd
	101	PE	Parity Even
	110	P	Positive
	111	M	Negative

Table C.5: Repeat and Direction bits **C** and **D**

C	Repeat	D	Direction
0	Run Once	0	Increment address
		1	Decrement address
1	Repeat until $BC = 0$	0	Increment address
		1	Decrement address

This is used by the **CP⊠⊠**, **IN⊠⊠**, **LD⊠⊠** or **OUT⊠⊠** instruction families.

e.g. LDI, LDIR, OUTI or OTIR *etc.*

Table C.6: Bit number **b**

b	Bit Number
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

C.3.2 Decoding tables

Table C.7: Z80 Instruction Decode Table

		Len	Byte +0								Byte +1								Byte +2								Sect.	Page
				7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	
NOP		1	0	0	0	0	0	0	0	0																		22.36	336
LD	(BC),A	1	0	0	0	0	0	0	0	1	0																	22.30	320
RLCA		1	0	0	0	0	0	0	1	1	1																	22.52	362
EX	AF,AF'	1	0	0	0	0	1	0	0	0																		22.18	302
LD	A,(BC)	1	0	0	0	0	1	0	1	0																		22.30	320
RRCA		1	0	0	0	0	1	1	1	1																		22.56	367
DJNZ	e	2	0	0	0	1	0	0	0	0	e-2																	22.16	299
LD	(DE),A	1	0	0	0	1	0	0	1	0																		22.30	320
RLA		1	0	0	0	1	0	1	1	1																		22.50	359
JR	e	2	0	0	0	1	1	0	0	0	e-2																	22.29	319
LD	A,(DE)	1	0	0	0	1	1	0	1	0																		22.30	320
RRA		1	0	0	0	1	1	1	1	1																		22.55	366
LD	(nn),HL	3	0	0	1	0	0	0	1	0	n									n								22.30	320
DAA		1	0	0	1	0	0	1	1	1																		22.13	294
LD	HL,(nn)	3	0	0	1	0	1	0	1	0	n									n								22.30	320
CPL		1	0	0	1	0	1	1	1	1																		22.12	293
LD	(nn),A	3	0	0	1	1	0	0	1	0	n									n								22.30	320
INC	(HL)	1	0	0	1	1	0	1	0	0																		22.23	310
DEC	(HL)	1	0	0	1	1	0	1	0	1																		22.14	296
LD	(HL),n	2	0	0	1	1	0	1	1	0	n																	22.30	320
SCF		1	0	0	1	1	0	1	1	1																		22.61	377
LD	A,(nn)	3	0	0	1	1	1	0	1	0	n									n								22.30	320
CCF		1	0	0	1	1	1	1	1	1																		22.6	285
JR	cc,nn	2	0	0	1	c			0	0	0	e-2																22.29	319
INC	r	1	0	0	r				1	0	0																	22.23	310
DEC	r	1	0	0	r				1	0	1																	22.14	296
LD	r,n	2	0	0	r				1	1	0	n																22.30	320
LD	ss,nn	3	0	0	s		0	0	0	1	n									n								22.30	320
INC	ss	1	0	0	s		0	0	1	1																		22.23	310
ADD	HL,ss	1	0	0	s		1	0	0	1																		22.2	278
DEC	ss	1	0	0	s		1	0	1	1																		22.14	296
HALT		1	0	1	1	1	0	1	1	0																		22.20	305
LD	(HL),r	1	0	1	1	1	0	r																				22.30	320
LD	r,(HL)	1	0	1	r			1	1	0																		22.30	320
LD	r,r'	1	0	1	r			r'																				22.30	320
ADD	A,(HL)	1	1	0	0	0	0	1	1	0																		22.2	278
ADD	A,r	1	1	0	0	0	0	r																				22.2	278
ADC	A,(HL)	1	1	0	0	0	1	1	1	0																		22.1	276
ADC	A,r	1	1	0	0	0	1	r																				22.1	276
SUB	(HL)	1	1	0	0	1	0	1	1	0																		22.66	388
SUB	r	1	1	0	0	1	0	r																				22.66	388
SBC	(HL)	1	1	0	0	1	1	1	1	0																		22.60	375

Table C.7: Z80 Instruction Decode Table (continued)

	Len	Byte +0								Byte +1								Byte +2								Sect.	Page
		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
SBC	r	1	1	0	0	1	1	r																		22.60	375
AND	(HL)	1	1	0	1	0	0	1	0																	22.3	280
AND	r	1	1	0	1	0	0	r																		22.3	280
XOR	(HL)	1	1	0	1	0	1	1	0																	22.67	390
XOR	r	1	1	0	1	0	1	r																		22.67	390
OR	(HL)	1	1	0	1	1	0	1	0																	22.37	337
OR	r	1	1	0	1	1	0	r																		22.37	337
CP	(HL)	1	1	0	1	1	1	1	0																	22.7	286
CP	r	1	1	0	1	1	1	r																		22.7	286
JP	nn	3	1	1	0	0	0	0	1	1	a								a							22.28	317
ADD	A,n	2	1	1	0	0	0	1	1	0	n															22.2	278
RET		1	1	1	0	0	1	0	0	1																22.46	354
CALL	nn	3	1	1	0	0	1	1	0	1	a								a							22.5	283
ADC	A,n	2	1	1	0	0	1	1	1	0	n															22.1	276
OUT	(N),A	2	1	1	0	1	0	0	1	1	n															22.40	341
SUB	n	2	1	1	0	1	0	1	1	0	n															22.66	388
EXX		1	1	1	0	1	1	0	0	1																22.19	304
IN	A,(N)	2	1	1	0	1	1	0	1	1	a															22.22	308
SBC	n	2	1	1	0	1	1	1	1	0	n															22.60	375
AND	n	2	1	1	1	0	0	1	1	0	n															22.3	280
JP	(HL)	1	1	1	1	0	1	0	0	1																22.28	317
EX	DE,HL	1	1	1	1	0	1	0	1	1																22.18	302
EX	(SP),HL	1	1	1	1	0	1	1	0	0																22.18	302
XOR	n	2	1	1	1	0	1	1	1	0	n															22.67	390
DI		1	1	1	1	1	0	0	1	1																22.15	298
OR	n	2	1	1	1	1	0	1	1	0	n															22.37	337
LD	SP,HL	1	1	1	1	1	1	0	0	1																22.30	320
EI		1	1	1	1	1	1	0	1	1																22.17	301
CP	n	2	1	1	1	1	1	1	1	0	n															22.7	286
RET	c	1	1	1	c			0	0	0																22.46	354
JP	cc,nn	3	1	1	c			0	1	0	a								a							22.28	317
CALL	cc,nn	3	1	1	c			1	0	0	a								a							22.5	283
RST		1	1	1	p			1	1	1																22.59	371
POP	qq	1	1	1	q		0	0	0	1																22.43	346
PUSH	qq	1	1	1	q		0	1	0	1																22.44	348

Table C.8: Z80 Instruction Decode Table - CB PREFIX

Byte 0 is always `0xCB`

	Len	Byte +1								Byte +2								Byte +3								Sect.	Page
		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
RLC	(HL)	2	0	0	0	0	0	1	1	0																22.51	360
RLC	r	2	0	0	0	0	0	r																		22.51	360
RRC	(HL)	2	0	0	0	0	1	1	1	0																22.57	368
RRC	r	2	0	0	0	0	1	r																		22.57	368

Table C.8: Z80 Instruction Decode Table - CB PREFIX (continued)

Byte 0 is always `0xCB`

		Len	Byte +1								Byte +2								Byte +3								Sect.	Page
			7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
RL	(HL)	2	0	0	0	0	1	0	1	0																	22.49	357
RL	r	2	0	0	0	0	1	0	r																		22.49	357
RR	(HL)	2	0	0	0	0	1	1	1	0																	22.54	364
RR	r	2	0	0	0	0	1	1	r																		22.54	364
SLA	(HL)	2	0	0	0	1	0	0	1	0																	22.63	382
SLA	r	2	0	0	0	1	0	0	r																		22.63	382
SRA	(HL)	2	0	0	0	1	0	1	1	0																	22.64	384
SRA	r	2	0	0	0	1	0	1	r																		22.64	384
SRL	(HL)	2	0	0	0	1	1	1	1	0																	22.65	386
SRL	r	2	0	0	0	1	1	1	r																		22.65	386
BIT	b,(HL)	2	0	1	b			1	1	0																	22.4	281
BIT	b,r	2	0	1	b			r																			22.4	281
RES	b,(HL)	2	1	0	b			1	1	0																	22.45	350
RES	b,r	2	1	0	b			r																			22.45	350
SET	(HL)	2	1	1	b			1	1	0																	22.62	378
SET	r	2	1	1	b			r																			22.62	378

Table C.9: Z80 Instruction Decode Table - DD PREFIX

Byte 0 is always `0xDD`

		Len	Byte +1								Byte +2								Byte +3								Sect.	Page
			7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
LD	IX,n	4	0	0	1	0	0	0	0	1	n								n								22.30	320
LD	(nn),IX	4	0	0	1	0	0	0	1	0	n								n								22.30	320
INC	IX	2	0	0	1	0	0	0	1	1																	22.23	310
LD	IX,(nn)	4	0	0	1	0	1	0	1	0	n								n								22.30	320
DEC	IX	2	0	0	1	0	1	0	1	1																	22.14	296
INC	(IX+d)	3	0	0	1	1	0	1	0	0	d																22.23	310
DEC	(IX+d)	3	0	0	1	1	0	1	0	1	d																22.14	296
LD	(IX+d),n	4	0	0	1	1	0	1	1	0	d								n								22.30	320
LD	(IX+d),r	3	0	1	1	1	0	r			d																22.30	320
LD	r,(IX+d)	3	0	1	r			1	1	0	d																22.30	320
ADD	A,(IX+d)	3	1	0	0	0	0	1	1	0	d																22.2	278
ADC	A,(IX+d)	3	1	0	0	0	1	1	1	0	d																22.1	276
SUB	(IX+d)	3	1	0	0	1	0	1	1	0	d																22.66	388
SBC	(IX+d)	3	1	0	0	1	1	1	1	0	d																22.60	375
AND	(IX+d)	3	1	0	1	0	0	1	1	0	d																22.3	280
XOR	(IX+d)	3	1	0	1	0	1	1	1	0	d																22.67	390
OR	(IX+d)	3	1	0	1	1	0	1	1	0	d																22.37	337
CP	(IX+d)	3	1	0	1	1	1	1	1	0	d																22.7	286
RLC	(IX+d)	4	1	1	0	0	1	0	1	1	d								0	0	0	0	0	1	1	0	22.51	360
RRC	(IX+d)	4	1	1	0	0	1	0	1	1	d								0	0	0	0	1	1	1	0	22.57	368
RL	(IX+d)	4	1	1	0	0	1	0	1	1	d								0	0	0	1	0	1	1	0	22.49	357
RR	(IX+d)	4	1	1	0	0	1	0	1	1	d								0	0	0	1	1	1	1	0	22.54	364

Table C.9: Z80 Instruction Decode Table - DD PREFIX (continued)

Byte 0 is always `0xDD`

		Len	Byte +1								Byte +2								Byte +3								Sect.	Page
			7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
SLA	(IX+d)	4	1	1	0	0	1	0	1	1	d								0	0	1	0	0	1	1	0	22.63	382
SRA	(IX+d)	4	1	1	0	0	1	0	1	1	d								0	0	1	0	1	1	1	0	22.64	384
SRL	(IX+d)	4	1	1	0	0	1	0	1	1	d								0	0	1	1	1	1	1	0	22.65	386
BIT	b,(IX+d)	4	1	1	0	0	1	0	1	1	d								0	1	b			1	1	0	22.4	281
RES	b,(IX+d)	4	1	1	0	0	1	0	1	1	d								1	0	b			1	1	0	22.45	350
SET	(IX+d)	4	1	1	0	0	1	0	1	1	d								1	1	b			1	1	0	22.62	378
POP	IX	2	1	1	1	0	0	0	0	1																	22.43	346
PUSH	IX	2	1	1	1	0	0	1	0	1																	22.44	348
JP	(IX)	2	1	1	1	0	1	0	0	1																	22.28	317
EX	(SP),IX	2	1	1	1	0	1	1	0	0																	22.18	302
LD	SP,IX	2	1	1	1	1	1	0	0	1																	22.30	320

Table C.10: Z80 Instruction Decode Table - ED PREFIX

Byte 0 is always `0xED`

		Len	Byte +1								Byte +2								Byte +3								Sect.	Page
			7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
SBC	HL,ss	2	0	1	s		0	0	1	0																	22.60	375
NEG		2	0	1	0	0	0	0	1	0																	22.35	335
RETN		2	0	1	0	0	0	0	1	0																	22.48	356
IM	0	2	0	1	0	0	0	1	1	0																	22.21	306
LD	I,A	2	0	1	0	0	0	1	1	1																	22.30	320
RETI		2	0	1	0	0	1	1	0	1																	22.47	355
LD	R,A	2	0	1	0	0	1	1	1	1																	22.30	320
IM	1	2	0	1	0	1	0	1	1	0																	22.21	306
LD	A,I	2	0	1	0	1	0	1	1	1																	22.30	320
IM	2	2	0	1	0	1	1	1	1	0																	22.21	306
LD	A,R	2	0	1	0	1	1	1	1	1																	22.30	320
RRD		2	0	1	1	0	0	1	1	1																	22.58	370
RLD		2	0	1	1	0	1	1	1	1																	22.53	363
IN	r,(C)	2	0	1	r			0	0	0																	22.22	308
OUT	(C),r	2	0	1	r			0	0	1																	22.40	341
LD	(nn),ss	4	0	1	s		0	0	1	1	n								n								22.30	320
ADC	HL,ss	2	0	1	s		1	0	1	0																	22.1	276
LD	ss,(nn)	4	0	1	s		1	0	1	1	n								n								22.30	320
LDᵢᵣ		2	1	0	1	C	D	0	0	0																	22.31	330
CPᵢᵣ		2	1	0	1	C	D	0	0	1																	22.8	288
INᵢᵣ		2	1	0	1	C	D	0	1	0																	22.24	312
OUTᵢᵣ		2	1	0	1	C	D	0	1	1																	22.41	343

Table C.11: Z80 Instruction Decode Table - FD PREFIX

Byte 0 is always `0xFD`

		Len	Byte +1								Byte +2								Byte +3								Sect.	Page
			7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
LD	IY,n	4	0	0	1	0	0	0	0	1	n								n								22.30	320
LD	(nn),IY	4	0	0	1	0	0	0	1	0	n								n								22.30	320
INC	IY	2	0	0	1	0	0	0	1	1																	22.23	310
LD	IY,(nn)	4	0	0	1	0	1	0	1	0	n								n								22.30	320
DEC	IY	2	0	0	1	0	1	0	1	1																	22.14	296
INC	(IY+d)	3	0	0	1	1	0	1	0	0	d																22.23	310
DEC	(IY+d)	3	0	0	1	1	0	1	0	1	d																22.14	296
LD	(IY+d),n	4	0	0	1	1	0	1	1	0	d								n								22.30	320
LD	r,(IY+d)	3	0	1	r			1	1	0	d																22.30	320
ADD	A,(IY+d)	3	1	0	0	0	0	1	1	0	d																22.2	278
ADC	A,(IY+d)	3	1	0	0	0	1	1	1	0	d																22.1	276
SUB	A,(IY+d)	3	1	0	0	1	0	1	1	0	d																22.66	388
SBC	(IY+d)	3	1	0	0	1	1	1	1	0	d																22.60	375
AND	(IY+d)	3	1	0	1	0	0	1	1	0	d																22.3	280
XOR	A,(IY+d)	3	1	0	1	0	1	1	1	0	d																22.67	390
OR	(IY+d)	3	1	0	1	1	0	1	1	0	d																22.37	337
CP	(IY+d)	3	1	0	1	1	1	1	1	0	d																22.7	286
RLC	(IY+d)	4	1	1	0	0	1	0	1	1	d								0	0	0	0	0	1	1	0	22.51	360
RRC	(IY+d)	4	1	1	0	0	1	0	1	1	d								0	0	0	0	1	1	1	0	22.57	368
RL	(IY+d)	4	1	1	0	0	1	0	1	1	d								0	0	0	1	0	1	1	0	22.49	357
RR	(IY+d)	4	1	1	0	0	1	0	1	1	d								0	0	0	1	1	1	1	0	22.54	364
SLA	(IY+d)	4	1	1	0	0	1	0	1	1	d								0	0	1	0	0	1	1	0	22.63	382
SRA	(IY+d)	4	1	1	0	0	1	0	1	1	d								0	0	1	0	1	1	1	0	22.64	384
SRL	(IY+d)	4	1	1	0	0	1	0	1	1	d								0	0	1	1	1	1	1	0	22.65	386
BIT	b,(IY+d)	4	1	1	0	0	1	0	1	1	d								0	1	b			1	1	0	22.4	281
RES	b,(IY+d)	4	1	1	0	0	1	0	1	1	d								1	0	b			1	1	0	22.45	350
SET	(IY+d)	4	1	1	0	0	1	0	1	1	d								1	1	b			1	1	0	22.62	378
POP	IY	2	1	1	1	0	0	0	0	1																	22.43	346
PUSH	IY	2	1	1	1	0	0	1	0	1																	22.44	348
JP	(IY)	2	1	1	1	0	1	0	0	1																	22.28	317
EX	(SP),IY	2	1	1	1	0	1	1	0	0																	22.18	302
LD	SP,IY	2	1	1	1	1	1	0	0	1																	22.30	320

Page intentionally blank

Acorn Machine Operating System D

D.1 Calls and Vectors

Table D.1 lists the entry points to the main MOS and the indirection vectors for those functions.

Table D.1: MOS System Calls

Name	Addr	Vector	Addr	Function	Sect
OSWRSC	FFB3			Write byte to screen	
OSRDSC	FFB9			Read byte from screen	
OSEVEN	FFBF			Generate an event	30.8
GSINIT	FFC2			String input initialize	
GSREAD	FFC5			Read char from string	
NVRDCH	FFC8			Non-vectored read char	30.4
NVWRCH	FFCB			Non-vectored write char	30.1
OSFIND	FFCE	FINDV	021C	Open or Close file	
OSGBPB	FFD1	GBPBV	021A	Load or Save memory to file	
OSBPUT	FFD4	BPUTV	0218	Save a single byte to file	30.6
OSBGET	FFD7	BGETV	0216	Read a single byte from file	30.5
OSARGS	FFDA	ARGSV	0214	Load or Save data about a file	
OSFILE	FFDD	FILEV	0212	Load or Save a complete file	
OSRDCH	FFE0	RDCHV	0210	Read character to A	30.4
OSASCI	FFE3			Write char to screen, add LF if CR	30.3
OSNEWL	FFE7			Write LF then CR to screen	30.2
OSWRCH	FFEE	WRCHV	020E	Write character in A to screen	30.1
OSWORD	FFF1	WORDV	020C	Perform OSWORD call	
OSBYTE	FFF4	BYTEV	020A	Perform OSBYTE call	30.10
OSCLI	FFF7	CLIV	0208	Interpret command line	30.7

D.1.1 Errata

The BBC B User Guide Appendix O[10] shows BPUTV address as 2 when it should read 218.

[10] *BBC Micro. User Guide*, (Oct. 1984) P.506

D.2 OSByte calls

This section lists the documented **OSBYTE** calls described in Section 30.10.

D.2.1 General OSBYTE calls

Table D.2 lists **OSBYTE** calls `0x00`...`A5` which provide specific actions.

Table D.3: P.970

Table D.3 lists them by the action name.

D.2.2 System Variable OSBYTE calls

Table D.8: P.978

Table D.8 shows the **OSBYTE** calls `0xA6`...`FF` which provide read write access to the MOS variables.

Table D.9: P.980

Table D.9 lists them by the variable name.

D.2.3 Table layout

In both sets of tables, the platform columns show which platforms support the list call.

Not all platforms support a call, but where it is either documented as supported – or issues a call to Paged ROMs, then it will be shown with a ⋆.

Some calls have been redefined between platforms, especially those which work on the MOS variables, so they are shown as separate entries.

Entries which are greyed out simply shows that a specific **OSBYTE** call differs between platforms.

Table D.2: OSByte calls

Dec	Hex	Action	BBC B	BBC B+	Electron	Master	Compact
0	00	Get OS Version	★	★	★	★	★
1	01	Write user flag	★	★	★	★	★
2	02	Specify input stream	★	★	★	★	★
3	03	Specify output stream	★	★	★	★	★
4	04	Enable/disable cursor editing	★	★	★	★	★
5	05	Select printer destination	★	★	★	★	★
6	06	Write printer ignore character	★	★	★	★	★
7	07	Set RS-423 receive rate	★	★	★	★	★
8	08	Set RS-423 transmit rate	★	★	★	★	★
9	09	Set duration of first colour	★	★	★	★	★
10	0A	Write duration of second colour	★	★	★	★	★
11	0B	Set keyboard auto-repeat delay	★	★	★	★	★
12	0C	Set keyboard auto-repeat rate	★	★	★	★	★
13	0D	Disable event	★	★	★	★	★
14	0E	Enable event	★	★	★	★	★
15	0F	Flush buffers	★	★	★	★	★
16	10	Select ADC channel	★	★	★	★	★
17	11	Force ADC conversion	★	★	★	★	★
18	12	Reset soft function keys	★	★	★	★	★
19	13	Wait for vertical sync	★	★	★	★	★
20	14	Explode character definition RAM	★	★	★		
20	*14*	Reset character definitions				★	★
21	15	Flush specific buffer	★	★	★	★	★
22	16	Increment polling semaphore			★	★	★
23	17	Decrement polling semaphore			★	★	★
24	18	Select external Sound			★		
25	19	Restore default font definitions				★	★
68	44	Test for sideways RAM		★		★	★
69	45	Sideways RAM allocation		★		★	★
107	6B	Select 1MHz bus/cartridge				★	
108	6C	Select screen memory for direct access				★	★
109	6D	Make temporary file system permanent				★	★
112	70	Select memory for VDU				★	★
113	71	Select memory for display				★	★
114	72	Write shadow memory use		★		★	★

Table D.2: OSByte calls (continued)

Dec	Hex	Action	BBC B	BBC B+	Electron	Master	Compact
115	73	Blank or restore palette			⋆		
116	74	Reset sound system			⋆		
117	75	Read VDU status	⋆	⋆	⋆	⋆	⋆
118	76	Make LEDs reflect keyboard status	⋆	⋆	⋆	⋆	⋆
119	77	Close *SPOOL/*EXEC files	⋆	⋆	⋆	⋆	⋆
120	78	Write current keys pressed	⋆	⋆	⋆	⋆	⋆
121	79	Perform keyboard scan	⋆	⋆	⋆	⋆	⋆
122	7A	Perform keyboard scan from 0x10	⋆	⋆	⋆	⋆	⋆
123	7B	Printer driver going dormant	⋆	⋆	⋆	⋆	⋆
124	7C	Clear ESCAPE condition	⋆	⋆	⋆	⋆	⋆
125	7D	Set ESCAPE condition	⋆	⋆	⋆	⋆	⋆
126	7E	Acknowledge ESCAPE, clear + effects	⋆	⋆	⋆	⋆	⋆
127	7F	Check for EOF	⋆	⋆	⋆	⋆	⋆
128	80	Read ADC channel/buffer status	⋆	⋆	⋆	⋆	⋆
129	81	Read key with time limit	⋆	⋆	⋆	⋆	⋆
129	*81*	Scan for specific key	⋆	⋆	⋆	⋆	⋆
129	*81*	Read machine type	⋆	⋆	⋆	⋆	⋆
130	82	Read machine high order address	⋆	⋆	⋆	⋆	⋆
131	83	Read top of OS RAM	⋆	⋆	⋆	⋆	⋆
132	84	Read top of user RAM	⋆	⋆	⋆	⋆	⋆
133	85	Read HIMEM for given mode	⋆	⋆	⋆	⋆	⋆
134	86	Read input cursor pos	⋆	⋆	⋆	⋆	⋆
135	87	Read char at text cursor and screen mode	⋆	⋆	⋆	⋆	⋆
136	88	*CODE	⋆	⋆	⋆	⋆	⋆
137	89	*MOTOR	⋆	⋆	⋆	⋆	⋆
138	8A	Insert into buffer	⋆	⋆	⋆	⋆	⋆
139	8B	*OPT	⋆	⋆	⋆	⋆	⋆
140	8C	*TAPE	⋆	⋆	⋆	⋆	⋆
141	8D	*ROM	⋆	⋆	⋆	⋆	⋆
142	8E	Enter language ROM	⋆	⋆	⋆	⋆	⋆
143	8F	Issue paged ROM service request	⋆	⋆	⋆	⋆	⋆
144	90	*TV	⋆	⋆		⋆	⋆
144	*90*	*TV			⋆		
145	91	Get character from buffer	⋆	⋆	⋆	⋆	⋆
146	92	Read from FRED	⋆	⋆	⋆	⋆	⋆

Table D.2: OSByte calls (continued)

Dec	Hex	Action	BBC B	BBC B+	Electron	Master	Compact
147	93	Write to FRED	⋆	⋆	⋆	⋆	⋆
148	94	Read from JIM	⋆	⋆	⋆	⋆	⋆
149	95	Write to JIM	⋆	⋆	⋆	⋆	⋆
150	96	Read from SHIELA	⋆	⋆	⋆	⋆	⋆
151	97	Write to SHIELA	⋆	⋆	⋆	⋆	⋆
152	98	Examine buffer status		⋆	⋆	⋆	⋆
152	*98*	Examine buffer status	⋆				
153	99	Insert character into input buffer	⋆	⋆	⋆	⋆	⋆
154	9A	Write to video ULA	⋆	⋆		⋆	⋆
154	*9A*	Reset flash cycle			⋆		
155	9B	Write to video ULA palette register	⋆	⋆		⋆	⋆
155	*9B*	No operation			⋆		
156	9C	Read/Write 6850 control register	⋆	⋆	⋆	⋆	⋆
157	9D	Fast Tube BPUT	⋆	⋆	⋆	⋆	⋆
158	9E	Read from speech processor	⋆	⋆	⋆		
158	*9E*	Reserved				⋆	⋆
159	9F	Write to speech processor	⋆	⋆	⋆		
159	*9F*	Reserved				⋆	⋆
160	A0	Read VDU variable		⋆	⋆	⋆	⋆
161	A1	Read CMOS RAM				⋆	
161	*A1*	Read EEPROM					⋆
161	*A1*	Query EEPROM size					⋆
162	A2	Write CMOS RAM/EEPROM				⋆	⋆
163	A3	Enable or Disable printer and ADC			⋆		
163	*A3*	Reserved for application software				⋆	⋆
164	A4	Check for 6502 code				⋆	⋆
165	A5	Read output cursor position				⋆	⋆

See Table D.8 for **OSBYTE** calls with 0xA6...FF

Table D.3: OSByte calls by action

Dec	Hex	Action	BBC B	BBC B+	Electron	Master	Compact
126	7E	Acknowledge ESCAPE, clear + effects	⋆	⋆	⋆	⋆	⋆
115	73	Blank or restore palette			⋆		
164	A4	Check for 6502 code				⋆	⋆
127	7F	Check for EOF	⋆	⋆	⋆	⋆	⋆
124	7C	Clear ESCAPE condition	⋆	⋆	⋆	⋆	⋆
119	77	Close *SPOOL/*EXEC files	⋆	⋆	⋆	⋆	⋆
23	17	Decrement polling semaphore			⋆	⋆	⋆
13	0D	Disable event	⋆	⋆	⋆	⋆	⋆
14	0E	Enable event	⋆	⋆	⋆	⋆	⋆
163	A3	Enable or Disable printer and ADC			⋆		
4	04	Enable/disable cursor editing	⋆	⋆	⋆	⋆	⋆
142	8E	Enter language ROM	⋆	⋆	⋆	⋆	⋆
152	98	Examine buffer status		⋆	⋆	⋆	⋆
152	*98*	Examine buffer status	⋆				
20	14	Explode character definition RAM	⋆	⋆	⋆		
157	9D	Fast Tube BPUT	⋆	⋆	⋆	⋆	⋆
15	0F	Flush buffers	⋆	⋆	⋆	⋆	⋆
21	15	Flush specific buffer	⋆	⋆	⋆	⋆	⋆
17	11	Force ADC conversion	⋆	⋆	⋆	⋆	⋆
145	91	Get character from buffer	⋆	⋆	⋆	⋆	⋆
0	00	Get OS Version	⋆	⋆	⋆	⋆	⋆
22	16	Increment polling semaphore			⋆	⋆	⋆
153	99	Insert character into input buffer	⋆	⋆	⋆	⋆	⋆
138	8A	Insert into buffer	⋆	⋆	⋆	⋆	⋆
143	8F	Issue paged ROM service request	⋆	⋆	⋆	⋆	⋆
118	76	Make LEDs reflect keyboard status	⋆	⋆	⋆	⋆	⋆
109	6D	Make temporary file system permanent				⋆	⋆
155	9B	No operation			⋆		
121	79	Perform keyboard scan	⋆	⋆	⋆	⋆	⋆
122	7A	Perform keyboard scan from 0x10	⋆	⋆	⋆	⋆	⋆
123	7B	Printer driver going dormant	⋆	⋆	⋆	⋆	⋆
161	A1	Query EEPROM size					⋆
128	80	Read ADC channel/buffer status	⋆	⋆	⋆	⋆	⋆
135	87	Read char at text cursor and screen mode	⋆	⋆	⋆	⋆	⋆
161	A1	Read CMOS RAM				⋆	

Table D.3: OSByte calls by action (continued)

Dec	Hex	Action	BBC B	BBC B+	Electron	Master	Compact
146	92	Read from FRED	⋆	⋆	⋆	⋆	⋆
148	94	Read from JIM	⋆	⋆	⋆	⋆	⋆
150	96	Read from SHIELA	⋆	⋆	⋆	⋆	⋆
158	9E	Read from speech processor	⋆	⋆	⋆		
133	85	Read HIMEM for given mode	⋆	⋆	⋆	⋆	⋆
134	86	Read input cursor pos	⋆	⋆	⋆	⋆	⋆
129	81	Read key with time limit	⋆	⋆	⋆	⋆	⋆
130	82	Read machine high order address	⋆	⋆	⋆	⋆	⋆
129	81	Read machine type	⋆	⋆	⋆	⋆	⋆
165	A5	Read output cursor position				⋆	⋆
131	83	Read top of OS RAM	⋆	⋆	⋆	⋆	⋆
132	84	Read top of user RAM	⋆	⋆	⋆	⋆	⋆
117	75	Read VDU status	⋆	⋆	⋆	⋆	⋆
160	A0	Read VDU variable		⋆	⋆	⋆	⋆
161	A1	Read EEPROM					⋆
156	9C	Read/Write 6850 control register	⋆	⋆	⋆	⋆	⋆
158	9E	Reserved				⋆	⋆
159	9F	Reserved				⋆	⋆
163	A3	Reserved for application software				⋆	⋆
20	14	Reset character definitions				⋆	⋆
154	9A	Reset flash cycle			⋆		
18	12	Reset soft function keys	⋆	⋆	⋆	⋆	⋆
116	74	Reset sound system			⋆		
25	19	Restore default font definitions				⋆	⋆
129	81	Scan for specific key	⋆	⋆	⋆	⋆	⋆
107	6B	Select 1MHz bus/cartridge				⋆	
16	10	Select ADC channel	⋆	⋆	⋆	⋆	⋆
24	18	Select external Sound			⋆		
113	71	Select memory for display				⋆	⋆
112	70	Select memory for VDU				⋆	⋆
5	05	Select printer destination	⋆	⋆	⋆	⋆	⋆
108	6C	Select screen memory for direct access				⋆	⋆
9	09	Set duration of first colour	⋆	⋆	⋆	⋆	⋆
125	7D	Set ESCAPE condition	⋆	⋆	⋆	⋆	⋆
11	0B	Set keyboard auto-repeat delay	⋆	⋆	⋆	⋆	⋆

Table D.3: OSByte calls by action (continued)

Dec	Hex	Action	BBC B	BBC B+	Electron	Master	Compact
12	0C	Set keyboard auto-repeat rate	⋆	⋆	⋆	⋆	⋆
7	07	Set RS-423 receive rate	⋆	⋆	⋆	⋆	⋆
8	08	Set RS-423 transmit rate	⋆	⋆	⋆	⋆	⋆
69	45	Sideways RAM allocation		⋆		⋆	⋆
2	02	Specify input stream	⋆	⋆	⋆	⋆	⋆
3	03	Specify output stream	⋆	⋆	⋆	⋆	⋆
68	44	Test for sideways RAM		⋆		⋆	⋆
19	13	Wait for vertical sync	⋆	⋆	⋆	⋆	⋆
162	A2	Write CMOS RAM/EEPROM				⋆	⋆
120	78	Write current keys pressed	⋆	⋆	⋆	⋆	⋆
10	0A	Write duration of second colour	⋆	⋆	⋆	⋆	⋆
6	06	Write printer ignore character	⋆	⋆	⋆	⋆	⋆
114	72	Write shadow memory use		⋆		⋆	⋆
147	93	Write to FRED	⋆	⋆	⋆	⋆	⋆
149	95	Write to JIM	⋆	⋆	⋆	⋆	⋆
151	97	Write to SHIELA	⋆	⋆	⋆	⋆	⋆
159	9F	Write to speech processor	⋆	⋆	⋆		
154	9A	Write to video ULA	⋆	⋆		⋆	⋆
155	9B	Write to video ULA palette register	⋆	⋆		⋆	⋆
1	01	Write user flag	⋆	⋆	⋆	⋆	⋆
136	88	*CODE	⋆	⋆	⋆	⋆	⋆
137	89	*MOTOR	⋆	⋆	⋆	⋆	⋆
139	8B	*OPT	⋆	⋆	⋆	⋆	⋆
141	8D	*ROM	⋆	⋆	⋆	⋆	⋆
140	8C	*TAPE	⋆	⋆	⋆	⋆	⋆
144	90	*TV	⋆	⋆		⋆	⋆
144	*90*	*TV			⋆		

See Table D.8 for **OSBYTE** calls with 0xA6...FF

D.3 OSWord calls

This section lists the documented **OSWORD** calls described in Section 30.11.

Table D.4: OSWord calls

Dec	Hex	Action	BBC B	BBC B+	Electron	Master	Compact
0	00	Read line from current input stream	⋆	⋆	⋆	⋆	⋆
1	01	Read system clock	⋆	⋆	⋆	⋆	⋆
2	02	Write system clock	⋆	⋆	⋆	⋆	⋆
3	03	Read interval timer	⋆	⋆	⋆	⋆	⋆
4	04	Write interval timer	⋆	⋆	⋆	⋆	⋆
5	05	Read byte of I/O processor memory	⋆	⋆	⋆	⋆	⋆
6	06	Write byte of I/O processor memory	⋆	⋆	⋆	⋆	⋆
7	07	Perform a SOUND command	⋆	⋆	⋆	⋆	⋆
8	08	Define an ENVELOPE	⋆	⋆	⋆	⋆	⋆
9	09	Read pixel value	⋆	⋆	⋆	⋆	⋆
10	0A	Read character definition	⋆	⋆	⋆	⋆	⋆
11	0B	Read palette for given logical colour	⋆	⋆	⋆	⋆	⋆
12	0C	Write palette for given logical colour	⋆	⋆	⋆	⋆	⋆
13	0D	Read previous and current graphics cursor position	⋆	⋆	⋆	⋆	⋆
14	0E	Read real time clock				⋆	
14	0E	Read real time clock					⋆
15	0F	Write real time clock				⋆	
255	FF	Read/Write I/O processor memory Z80	⋆	⋆	⋆	⋆	⋆

D.4 Zero-page use

Table D.5: P.975

Table D.5 shows how the 6502 zero-page is utilized under MOS.

D.4.1 Current Language

MOS reserves addresses `0x00`…`8F` to the current language.

When BBC Basic is the active language, it uses addresses `0x00`…`6F` for itself. It leaves `0x70`…`8F` free for user programs.

D.4.2 Econet Workspace

MOS reserves addresses `0x90`…`9F` to Econet.

D.4.3 NMI Workspace

MOS reserves addresses `0xa0`…`A8` to the current NMI handler. This is free on unexpanded machines, but disk filesystems and Econet use this.

D.4.4 File System Workspace

MOS reserves addresses `0xB0`…`CF` to the current filesystem. Table D.5 shows the TAPE and ROM filesystem usage, but this will be different for other filesystems.

D.4.5 VDU Workspace

MOS reserves addresses `0xD0`…`E1` to the VDU subsystem. `0xE0`…`E1` are not used on the BBC Master 128 series.

Table D.5: MOS zero-page

Addr	Title	Len	Function	Table
0000	Current Language Workspace	144	Reserved	
0090	Econet Workspace	16	Econet	
00A0	NMI workspace	8	Interrupt	
00A8	OS temp workspace	8		
00B0	TAPE/ROM file address	4		
00B0	Filing System Workspace	32	FileSys	
00B4	Current block number	2	TFS/RFS	
00B4	Save End Address	4	*TFS/RFS*	
00B6	Next block number	2	*TFS/RFS*	
00B8	Print message or err routine	2	*TFS/RFS*	
00BA	Current block error	1	*TFS/RFS*	
00BB	Tape options	1	*TFS/RFS*	D.6
00BC	Temp storage	1	*TFS/RFS*	
00BD	Char read or to write	1	*TFS/RFS*	
00BE	CRC Checksum	2	*TFS/RFS*	
00C0	B_7 set if char read/ready to write	1	*TFS/RFS*	
00C1	$\neq 0$ indicates error	1	*TFS/RFS*	
00C2	State reading tape, 0 ... 5	1	*TFS/RFS*	
00C3	File handow for current file	1	*TFS/RFS*	
00C4	Last Bput value	1	*TFS/RFS*	
00C5	Unused	1	*TFS/RFS*	
00C6	Tape Baud rate	1	*TFS/RFS*	
00C7	Inter-Block Gap	1	*TFS/RFS*	
00C8	OSFILE parameter block ptr	2	*TFS/RFS*	
00CA	Send to tape flag	1	*TFS/RFS*	
00CB	Counter when calculating CRC	1	*TFS/RFS*	
00CC	File length	2	*TFS/RFS*	
00CE	Unused	2	*TFS/RFS*	
00D0	Status byte	1	VDU	30.16
00D1	bits to write a pixel in a byte	1	*VDU*	
00D2	Text colour OR mask	1	*VDU*	
00D3	Text colour EOR mask	1	*VDU*	
00D4	Graphics colour OR mask	1	*VDU*	
00D5	Graphics colour EOR mask	1	*VDU*	
00D6	Graphics cursor address	2	*VDU*	
00D8	Text char cell address	2	*VDU*	
00DA	VDU Workspace	6	*VDU*	
00E0	Pointer to multiplication tables	2	*VDU*	
00E2	Filesystem Status Byte	1	FS	D.7
00E3	Tape options	1	TFS/RFS	D.6
00E4	Misc Workspace	3		
00E7	Autorepeat countdown timer	1	Keyboard	

Table D.5: MOS \glsentryname{zeropage} (continued)

Addr	Title	Len	Function	Table
00E8	Address of OSWORD 0 buffer	2		
00EA	RS-423 Timeout	1	RS-423	
00EB	Tape critical flag	1	TAPE FS	
00EC	Current keys pressed	2	Keyboard	29.1
00EE	Key ignored in OSBYTE 121/2	1	*Keyboard*	
00EF	Save A on OSBYTE/OSWORD	1		
00F0	Save X on OSBYTE/OSWORD	1		
00F1	Save Y on OSBYTE/OSWORD	1		
00F2	Address of input buffer	2		
00F4	Current selected ROM	1		
00F5	Speech ROM or ROMFS	1		
00F6	Rom Address	2		
00F8	Unused	2		
00FA	General purpose temp store	2		
00FC	Interrupt Accumulator	1	Interrupt	
00FD	Address after BRK	2	*Interrupt*	
00FD	Address of string to display	2		
00FF	Escape condition flag	1		

Table D.6: Tape filesystem options in 0x00BB

76 54	Message	Errors
00 00	None	Ignore
00 01	*None*	Abort
00 10	*None*	Retry
10 00	Short	Ignore
10 01	*Short*	Abort
10 10	*Short*	Retry
11 00	Long	Ignore
11 01	*Long*	Abort
11 10	*Long*	Retry

Bit	State
0	Input file open
1	output file open
2	Unused
3	current CATalogue status
4	Unused
5	Unused
6	EOF reached
7	EOF warning given

Table D.7: Filesystem Status byte

D.5 System Variables

The MOS system variables are stored in addresses `0x0236` ...`0x028F` on all machines. Table D.8 shows the MOS variables in address order, whilst Table D.9 is ordered by name.

Table D.8: P.978
Table D.9: P.980

D.5.1 Updating via OSBYTE

Each variable can be read and written to by a dedicated **OSBYTE** call starting from `0xA6`. The call number is shown in the OSByte column. Details on how to read and/or write the values are described in Subsection 30.10.1.

D.5.2 Table reference

The **Table** column shows the table reference which details the format of the variable if it is more than a simple value.

See Table D.2 for **OSBYTE** calls with `0x00`...`A5`

Table D.8: MOS Variables

Addr	Title	Len	BBC B	BBC B+	Electron	Master	Compact	OSByte	Table
0236	Base of MOS variables	2	⋆	⋆	⋆	⋆	⋆	166	A6
0238	ROM Pointer table	2	⋆	⋆	⋆	⋆	⋆	168	A8
023A	ROM Information table	2	⋆	⋆	⋆	⋆	⋆	170	AA
023C	Keyboard Translation Table	2	⋆	⋆	⋆	⋆	⋆	172	AC
023E	VDU Variables Table	2	⋆	⋆	⋆	⋆	⋆	174	AE
0240	CFS timeout counter	1	⋆	⋆	⋆	⋆	⋆	176	B0
0241	Input source	1	⋆	⋆	⋆	⋆	⋆	177	B1
0242	Keyboard semaphore	1	⋆	⋆	⋆	⋆	⋆	178	B2
0243	Default OSHWM	1	⋆	⋆	⋆			179	B3
0243	Paged ROM polling semaphore	1				⋆	⋆	179	B3
0244	Current OSHWM	1	⋆	⋆	⋆	⋆	⋆	180	B4
0245	RS-423 mode	1	⋆	⋆	⋆	⋆	⋆	181	B5
0246	Character Explosion State	1	⋆	⋆	⋆			182	B6
0246	Printer Ignore Character	1				⋆	⋆	182	B6
0247	TAPE or ROM Switch	1	⋆	⋆	⋆	⋆	⋆	183	B7
0248	Video ULA control register	1	⋆	⋆		⋆	⋆	184	B8
0249	Video ULA palette register	1	⋆	⋆		⋆	⋆	185	B9	30.25
0249	Paged ROM polling semaphore	1			⋆			185	B9
024A	ROM active on last BRK	1	⋆	⋆	⋆	⋆	⋆	186	BA
024B	BASIC ROM number	1	⋆	⋆	⋆	⋆	⋆	187	BB
024C	Current ADC channel being sampled	1	⋆	⋆	⋆	⋆	⋆	188	BC
024D	Maximum ADC channel number	1	⋆	⋆	⋆	⋆	⋆	189	BD
024E	ADC conversion type	1	⋆	⋆	⋆	⋆		190	BE	30.26
024E	ADC conversion type	1					⋆	190	BE	30.27
024F	RS-423 Use Flag	1	⋆	⋆	⋆	⋆	⋆	191	BF
0250	RS-423 6840 control register	1	⋆	⋆		⋆	⋆	192	C0
0250	RS-423 6840 control register	1			⋆			192	C0
0251	Read Flash Counter	1	⋆	⋆	⋆	⋆	⋆	193	C1
0252	Flash Mark Colour Duration	1	⋆	⋆	⋆	⋆	⋆	194	C2
0253	Flash Space Colour Duration	1	⋆	⋆	⋆	⋆	⋆	195	C3
0254	Keyboard Auto-repeat delay	1	⋆	⋆	⋆	⋆	⋆	196	C4
0255	Keyboard Auto-repeat rate	1	⋆	⋆	⋆	⋆	⋆	197	C5
0256	*EXEC File Handle	1	⋆	⋆	⋆	⋆	⋆	198	C6
0257	*SPOOL File Handle	1	⋆	⋆	⋆	⋆	⋆	199	C7
0258	Escape and Break Effect	1	⋆	⋆	⋆	⋆	⋆	200	C8	30.29
0259	Keyboard Disable	1	⋆	⋆	⋆	⋆	⋆	201	C9
025A	Keyboard Status	1	⋆	⋆	⋆	⋆	⋆	202	CA	30.30

Table D.8: MOS Variables (continued)

Addr	Title	Len	BBC B	BBC B+	Electron	Master	Compact	OSByte	Table
025B	RS-423 Handshake Level	1	⋆	⋆		⋆	⋆	203 CB	
025B	Electron ULA IRQ mask	1			⋆			203 CB	
025C	RS-423 Suppression Flag	1	⋆	⋆		⋆	⋆	204 CC	
025C	Firm Key Pointer	1			⋆			204 CC	
025D	RS-423 Selection Flag	1	⋆	⋆		⋆	⋆	205 CD	
025D	Firm Key string length	1			⋆			205 CD	
025E	Econet OS call interception status	1	⋆	⋆	⋆	⋆	⋆	206 CE	
025F	Econet read char interception status	1	⋆	⋆	⋆	⋆	⋆	207 CF	
0260	Econet write char interception status	1	⋆	⋆	⋆	⋆	⋆	208 D0	
0261	Speech suppression status	1	⋆	⋆	⋆	⋆	⋆	209 D1	30.24
0262	Sound suppression status	1	⋆	⋆	⋆	⋆	⋆	210 D2	
0263	Sound BELL channel	1	⋆	⋆	⋆	⋆	⋆	211 D3	
0264	Sound BELL amplitude/envelope	1	⋆	⋆	⋆	⋆	⋆	212 D4	
0265	Sound BELL pitch	1	⋆	⋆	⋆	⋆	⋆	213 D5	
0266	Sound BELL duration	1	⋆	⋆	⋆	⋆	⋆	214 D6	
0267	Startup message and Boot Options	1	⋆	⋆	⋆	⋆	⋆	215 D7	30.31
0268	Soft Key Length	1	⋆	⋆	⋆	⋆	⋆	216 D8	
0269	Lines printed since last page pause	1	⋆	⋆	⋆	⋆	⋆	217 D9	
026A	Number items in VDU queue	1	⋆	⋆	⋆	⋆	⋆	218 DA	
026B	TAB character	1	⋆	⋆		⋆	⋆	219 DB	
026B	External sound flag	1			⋆			219 DB	
026C	ESCape character	1	⋆	⋆	⋆	⋆	⋆	220 DC	
026D	Input buffer interp Cx	1	⋆	⋆	⋆	⋆	⋆	221 DD	30.32
026E	Input buffer interp Dx	1	⋆	⋆	⋆	⋆	⋆	222 DE	30.32
026F	Input buffer interp Ex	1	⋆	⋆	⋆	⋆	⋆	223 DF	30.32
0270	Input buffer interp Fx	1	⋆	⋆	⋆	⋆	⋆	224 E0	30.32
0271	Function Key status	1	⋆	⋆	⋆	⋆	⋆	225 E1	30.32
0272	Function Key + SHIFT status	1	⋆	⋆		⋆	⋆	226 E2	30.32
0272	Function Key + A...P status	1			⋆			226 E2	30.32
0273	Function Key + CTRL status	1	⋆	⋆		⋆	⋆	227 E3	30.32
0273	Function Key + Q...Z status	1			⋆			227 E3	30.32
0274	Function Key + CTRL SHIFT status	1	⋆	⋆		⋆	⋆	228 E4	30.32
0275	Escape action flag	1	⋆	⋆	⋆	⋆	⋆	229 E5	
0276	Escape effects flag	1	⋆	⋆	⋆	⋆	⋆	230 E6	
0277	User VIA IRQ bit mask	1	⋆	⋆		⋆	⋆	231 E7	
0277	Reserved for hardware expansion	1			⋆			231 E7	
0278	RS-423 6850 IRQ bit mask	1	⋆	⋆		⋆	⋆	232 E8	
0278	Sound semaphore	1			⋆			232 E8	
0279	System VIA IRQ bit mask	1	⋆	⋆		⋆	⋆	233 E9	

Table D.8: MOS Variables (continued)

Addr	Title	Len	BBC B	BBC B+	Electron	Master	Compact	OSByte	Table
0279	Soft key pointer	1			⋆			233 E9	
027A	Tube present flag	1	⋆	⋆	⋆	⋆	⋆	234 EA	
027B	Speech System present flag	1	⋆	⋆	⋆	⋆	⋆	235 EB	
027C	Output stream destination	1	⋆	⋆	⋆	⋆	⋆	236 EC	30.5
027D	Cursor editing status	1	⋆	⋆	⋆	⋆	⋆	237 ED	30.6
027E	Numeric Keypad base	1				⋆	⋆	238 EE	
027F	Shadow memory use	1		⋆		⋆	⋆	239 EF	
0280	Country Code	1			⋆	⋆	⋆	240 F0	30.33
0281	User Flag	1	⋆	⋆	⋆	⋆	⋆	241 F1	
0282	Serial ULA control register	1	⋆	⋆		⋆	⋆	242 F2	30.34
0282	Last value written to ULA	1			⋆			242 F2	
0283	Time Clock Switch	1	⋆	⋆	⋆	⋆	⋆	243 F3	
0284	Soft key consistency flag	1	⋆	⋆	⋆	⋆	⋆	244 F4	
0285	Printer destination	1	⋆	⋆	⋆	⋆	⋆	245 F5	30.7
0286	Printer ignore character	1	⋆	⋆	⋆	⋆	⋆	246 F6	
0287	Break Intercept	3	⋆	⋆	⋆	⋆	⋆	247 F7	
028A	Select memory for VDU	1				⋆	⋆	250 FA	
028B	Select memory for display	1				⋆	⋆	251 FB	
028C	Current language ROM	1	⋆	⋆	⋆	⋆	⋆	252 FC	
028D	Last reset type	1	⋆	⋆	⋆	⋆	⋆	253 FD	30.35
028E	RAM Size	1	⋆					254 FE	
028E	Reserved	1		⋆	⋆			254 FE	
028E	Keypad Shift Key Flag	1				⋆	⋆	254 FE	
028F	Startup Options	1	⋆	⋆	⋆	⋆	⋆	255 FF	30.37

See Table D.3 for **OSBYTE** calls with 0x00...A5 listed by name

Table D.9: MOS Variables by Name

Title	Address	Len	BBC B	BBC B+	Electron	Master	Compact	OSByte	Table
ADC conversion type	024E	1	⋆	⋆	⋆	⋆		190 BE	30.26
ADC conversion type	*024E*	1					⋆	190 BE	30.27
Base of MOS variables	0236	2	⋆	⋆	⋆	⋆	⋆	166 A6	
BASIC ROM number	024B	1	⋆	⋆	⋆	⋆	⋆	187 BB	
Break Intercept	0287	3	⋆	⋆	⋆	⋆	⋆	247 F7	
CFS timeout counter	0240	1	⋆	⋆	⋆	⋆	⋆	176 B0	

Table D.9: MOS Variables by Name (continued)

Title	Address	Len	BBC B	BBC B+	Electron	Master	Compact	OSByte	Table
Character Explosion State	0246	1	⋆	⋆	⋆			182 B6	
Country Code	0280	1			⋆	⋆	⋆	240 F0	30.33
Current ADC channel being sampled	024C	1	⋆	⋆	⋆	⋆	⋆	188 BC	
Current language ROM	028C	1	⋆	⋆	⋆	⋆	⋆	252 FC	
Current OSHWM	0244	1	⋆	⋆	⋆	⋆	⋆	180 B4	
Cursor editing status	027D	1	⋆	⋆	⋆	⋆	⋆	237 ED	30.6
Default OSHWM	0243	1	⋆	⋆	⋆			179 B3	
Econet OS call interception status	025E	1	⋆	⋆	⋆	⋆	⋆	206 CE	
Econet read char interception status	025F	1	⋆	⋆	⋆	⋆	⋆	207 CF	
Econet write char interception status	0260	1	⋆	⋆	⋆	⋆	⋆	208 D0	
Electron ULA IRQ mask	025B	1			⋆			203 CB	
Escape action flag	0275	1	⋆	⋆	⋆	⋆	⋆	229 E5	
Escape and Break Effect	0258	1	⋆	⋆	⋆	⋆	⋆	200 C8	30.29
ESCape character	026C	1	⋆	⋆	⋆	⋆	⋆	220 DC	
Escape effects flag	0276	1	⋆	⋆	⋆	⋆	⋆	230 E6	
External sound flag	026B	1			⋆			219 DB	
Firm Key Pointer	025C	1			⋆			204 CC	
Firm Key string length	025D	1			⋆			205 CD	
Flash Mark Colour Duration	0252	1	⋆	⋆	⋆	⋆	⋆	194 C2	
Flash Space Colour Duration	0253	1	⋆	⋆	⋆	⋆	⋆	195 C3	
Function Key status	0271	1	⋆	⋆	⋆	⋆	⋆	225 E1	30.32
Function Key + A...P status	0272	1			⋆			226 E2	30.32
Function Key + CTRL SHIFT status	0274	1	⋆	⋆		⋆	⋆	228 E4	30.32
Function Key + CTRL status	0273	1	⋆	⋆		⋆	⋆	227 E3	30.32
Function Key + Q...Z status	*0273*	1			⋆			227 E3	30.32
Function Key + SHIFT status	0272	1	⋆	⋆		⋆	⋆	226 E2	30.32
Input buffer interp Cx	026D	1	⋆	⋆	⋆	⋆	⋆	221 DD	30.32
Input buffer interp Dx	026E	1	⋆	⋆	⋆	⋆	⋆	222 DE	30.32
Input buffer interp Ex	026F	1	⋆	⋆	⋆	⋆	⋆	223 DF	30.32
Input buffer interp Fx	0270	1	⋆	⋆	⋆	⋆	⋆	224 E0	30.32
Input source	0241	1	⋆	⋆	⋆	⋆	⋆	177 B1	
Keyboard Auto-repeat delay	0254	1	⋆	⋆	⋆	⋆	⋆	196 C4	
Keyboard Auto-repeat rate	0255	1	⋆	⋆	⋆	⋆	⋆	197 C5	
Keyboard Disable	0259	1	⋆	⋆	⋆	⋆	⋆	201 C9	
Keyboard semaphore	0242	1	⋆	⋆	⋆	⋆	⋆	178 B2	
Keyboard Status	025A	1	⋆	⋆	⋆	⋆	⋆	202 CA	30.30
Keyboard Translation Table	023C	2	⋆	⋆	⋆	⋆	⋆	172 AC	
Keypad Shift Key Flag	028E	1				⋆	⋆	254 FE	
Last reset type	028D	1	⋆	⋆	⋆	⋆	⋆	253 FD	30.35

Table D.9: MOS Variables by Name (continued)

Title	Address	Len	BBC B	BBC B+	Electron	Master	Compact	OSByte	Table
Last value written to ULA	0282	1			⋆			242 F2	
Lines printed since last page pause	0269	1	⋆	⋆	⋆	⋆	⋆	217 D9	
Maximum ADC channel number	024D	1	⋆	⋆	⋆	⋆	⋆	189 BD	
Number items in VDU queue	026A	1	⋆	⋆	⋆	⋆	⋆	218 DA	
Numeric Keypad base	027E	1				⋆	⋆	238 EE	
Output stream destination	027C	1	⋆	⋆	⋆	⋆	⋆	236 EC	30.5
Paged ROM polling semaphore	0243	1				⋆	⋆	179 B3	
Paged ROM polling semaphore	0249	1			⋆			185 B9	
Printer destination	0285	1	⋆	⋆	⋆	⋆	⋆	245 F5	30.7
Printer Ignore Character	0246	1				⋆	⋆	182 B6	
Printer ignore character	0286	1	⋆	⋆	⋆	⋆	⋆	246 F6	
RAM Size	028E	1	⋆					254 FE	
Read Flash Counter	0251	1	⋆	⋆	⋆	⋆	⋆	193 C1	
Reserved	028E	1		⋆	⋆			254 FE	
Reserved for hardware expansion	0277	1			⋆			231 E7	
ROM active on last BRK	024A	1	⋆	⋆	⋆	⋆	⋆	186 BA	
ROM Information table	023A	2	⋆	⋆	⋆	⋆	⋆	170 AA	
ROM Pointer table	0238	2	⋆	⋆	⋆	⋆	⋆	168 A8	
RS-423 6840 control register	0250	1	⋆	⋆		⋆	⋆	192 C0	
RS-423 6840 control register	*0250*	1			⋆			192 C0	
RS-423 6850 IRQ bit mask	0278	1	⋆	⋆		⋆	⋆	232 E8	
RS-423 Handshake Level	025B	1	⋆	⋆		⋆	⋆	203 CB	
RS-423 mode	0245	1	⋆	⋆	⋆	⋆	⋆	181 B5	
RS-423 Selection Flag	025D	1	⋆	⋆		⋆	⋆	205 CD	
RS-423 Suppression Flag	025C	1	⋆	⋆		⋆	⋆	204 CC	
RS-423 Use Flag	024F	1	⋆	⋆	⋆	⋆	⋆	191 BF	
Select memory for display	028B	1				⋆	⋆	251 FB	
Select memory for VDU	028A	1				⋆	⋆	250 FA	
Serial ULA control register	0282	1	⋆	⋆		⋆	⋆	242 F2	30.34
Shadow memory use	027F	1		⋆		⋆	⋆	239 EF	
Soft key consistency flag	0284	1	⋆	⋆	⋆	⋆	⋆	244 F4	
Soft Key Length	0268	1	⋆	⋆	⋆	⋆	⋆	216 D8	
Soft key pointer	0279	1			⋆			233 E9	
Sound BELL amplitude/envelope	0264	1	⋆	⋆	⋆	⋆	⋆	212 D4	
Sound BELL channel	0263	1	⋆	⋆	⋆	⋆	⋆	211 D3	
Sound BELL duration	0266	1	⋆	⋆	⋆	⋆	⋆	214 D6	
Sound BELL pitch	0265	1	⋆	⋆	⋆	⋆	⋆	213 D5	
Sound semaphore	0278	1			⋆			232 E8	
Sound suppression status	0262	1	⋆	⋆	⋆	⋆	⋆	210 D2	

Table D.9: MOS Variables by Name (continued)

Title	Address	Len	BBC B	BBC B+	Electron	Master	Compact	OSByte	Table
Speech suppression status	0261	1	⋆	⋆	⋆	⋆	⋆	209 D1	30.24
Speech System present flag	027B	1	⋆	⋆	⋆	⋆	⋆	235 EB	
Startup message and Boot Options	0267	1	⋆	⋆	⋆	⋆	⋆	215 D7	30.31
Startup Options	028F	1	⋆	⋆	⋆	⋆	⋆	255 FF	30.37
System VIA IRQ bit mask	0279	1	⋆	⋆		⋆	⋆	233 E9	
TAB character	026B	1	⋆	⋆		⋆	⋆	219 DB	
TAPE or ROM Switch	0247	1	⋆	⋆	⋆	⋆	⋆	183 B7	
Time Clock Switch	0283	1	⋆	⋆	⋆	⋆	⋆	243 F3	
Tube present flag	027A	1	⋆	⋆	⋆	⋆	⋆	234 EA	
User Flag	0281	1	⋆	⋆	⋆	⋆	⋆	241 F1	
User VIA IRQ bit mask	0277	1	⋆	⋆		⋆	⋆	231 E7	
VDU Variables Table	023E	2	⋆	⋆	⋆	⋆	⋆	174 AE	
Video ULA control register	0248	1	⋆	⋆		⋆	⋆	184 B8	
Video ULA palette register	0249	1	⋆	⋆		⋆	⋆	185 B9	30.25
*EXEC File Handle	0256	1	⋆	⋆	⋆	⋆	⋆	198 C6	
*SPOOL File Handle	0257	1	⋆	⋆	⋆	⋆	⋆	199 C7	

D.6 Machine type returned by INKEY and OSBYTE 0x81

Table D.10 lists the possible results from either **OSBYTE `0x81`** or the BBC Basic **INKEY** function when querying the machine type. The table lists all of the possible values from BBC Basic including non-Acorn 8-bit platforms.

Table D.10: Machine type returned by INKEY and OSBYTE 0x81

Type	Machine Type	Variant	OS
`0x00`	BBC Model B		`0.10`
`0x00`	BBC Model A		`0.10`
`0x00`	RM Nimbus		
`0x00`	RunCPM	MS-Windows	
`0x01`	Acorn Electron		`1.00`
`0x01`	RunCPM	Arduino	
`0x02`	RunCPM	POSIX/Linux	
`0x03`	RunCPM	MS-Dos	
`0x04`	RunCPM	Teensy	

Table D.10: Machine type returned by INKEY and OSBYTE 0x81 (continued)

Type	Machine Type	Variant	OS
0x05	RunCPM	ESP32	
0x06	RunCPM	STM32	
0x2#	Apple II hardware		
0x2A	Apple IIgs		
0x2C	Apple IIc		
0x2E	Apple IIe		
0x4D	Matrix Brandy Basic	not RISC-OS	from 1.22.6
0x4D	Matrix Brandy Basic	including RISC-OS	from 1.22.12
0x50	Sega Master System		
0x53	BBC Basic on SDL	Assembly source	
0x57	BBC Basic	MS-Windows	
0x63	6809 system		
0x68	6309 system		
0x73	BBC Basic on SDL	C source	
0x8#	TI Calculator		
0x83	TI-83+ (SE)		
0x84	TI-84+ (SE)		
0xA#	ARM based systems		
0xA0	Arthur		1.20
0xA1	RISC-OS		2.00
0xA2	RISC-OS		2.01
0xA3	RISC-OS		3.0x
0xA4	RISC-OS		3.1x
0xA5	RISC-OS		3.50
0xA6	RISC-OS		3.60
0xA7	RISC-OS		3.7x
0xA8	RISC-OS		4.0x
0xA9	RISC-OS		4.3x
0xAA	RISC-OS		5.xx
0xAE	RISC-OS	Pyromaniac	
0xAF	Springboard		
0xB#	PDP-11		
0xB0	PDP-11	Unknown system	
0xB1	PDP-11	UKNC	
0xB5	PDP-11	Unix 5	
0xB6	PDP-11	Unix 6	
0xB7	PDP-11	Unix 7	
0xB9	PDP-11	BSD	2.9

Table D.10: Machine type returned by INKEY and OSBYTE 0x81 (continued)

Type	Machine Type	Variant	OS
0xBB	PDP-11	BSD	2.11
0xC#	Commodore hardware		
0xC4	C64		
0xC8	C128		
0xD#	Amstrad	CPC464, CPC6128 *etc.*	
0xE#	Spectrum hardware		
0xE0	Spectrum		
0xF3	GNU	Brandy Basic	
0xF4	GNU FreeBSD	Brandy Basic	
0xF5	BBC Master Compact		5.10
0xF5	Amiga	Brandy Basic	
0xF6	OpenBSD		
0xF7	FreeBSD		
0xF7	BBC Master Econet Terminal		4.00
0xF8	MacOS	Brandy Basic	
0xF9	Acorn Communicator		
0xF9	Linux		
0xFA	Acorn Business Computer		1.00
0xFA	Acorn Business Computer		2.00
0xFA	Cambridge Workstation		2.10
0xFB	BBC Model B+ 64K/128K		2.00
0xFB	BeOS	Brandy Basic	
0xFC	BBC Microcomputer	West Germany	1.20
0xFC	MS-Windows 32		
0xFD	BBC Master 128		3.20
0xFD	BBC Master 128		3.50
0xFE	BBC Microcomputer	USA	A1.00
0xFE	Reuters OS	*US	R0.3
0xFE	NetBSD		
0xFF	BBC Microcomputer		1.00
0xFF	BBC Microcomputer		1.20
0xFF	Reuters OS	*UK	R0.3

D.7 Keyboard key codes

Table D.11: P.988

Table D.11 lists the internal key code and the BBC Basic INKEY code for the keys on the keyboards for the BBC Model B, BBC Master 128 and Acorn Electron.

If a key is supported on a specific platform then it will have a ⋆ in the relevant system column.

See Chapter 29 for a full description on these codes.

D.7.1 Duplicate keys

Some keys are duplicated in the Table D.11 for various reasons. This is due to the table being generated from the same data as the keyboard layout diagrams, see Section 29.4.

Numeric Keypad

1: keypad Subsection 29.4.2

A ⋆ in the keypad column indicates that the key is on the numeric keypad[1] of the BBC Master 128 and BBC Master Compact, rather than on the main keyboard.

These keys share the ASCII value for the main keyboard equivalent, but are independent of the main keyboard, so they have unique internal and inkey codes.

Different key positions

Some other keys are duplicated for due to their positions being different between systems.

e.g. the Cursor and Copy keys vary in location between the three main keyboard types.

Different functionality

Some keys are duplicated as their functionality differs.

e.g. on the BBC Microcomputer, the '`0`' key is just that single character. On the Acorn Electron, it is shared with '`@`' and '`f0`'.

Caps Lock is another example. On the BBC Microcomputer it toggles the caps lock. However, on the Acorn Electron it is also the Function shift key if it's held down.

D.7.2 Break key

The Break[2] key is not included in Table D.11 as it's not part of the keyboard matrix. It is tied directly to the reset input of the 6502, so pressing it resets the machine.

2: Break Section 29.2

D.7.3 Shift keys

The two shift keys are shown as a single entry in Table D.11 as they are electrically identical in the keyboard matrix. Because of this, you cannot detect which one is being pressed.

Table D.11: BBC Micro Keyboard codes

		ASCII		Code		INKEY						
Key	Keypad	Hex	Dec	Hex	Dec	Hex	Dec	BBC B	BBC B+	Electron	Master	Compact
Tab		09	9	60	96	9F	-97	⋆	⋆		⋆	⋆
Return		0D	13	49	73	B6	-74	⋆	⋆	⋆	⋆	⋆
Return	⋆	0D	13	3C	60	C3	-61				⋆	⋆
Escape		1B	27	70	112	8F	-113	⋆	⋆	⋆	⋆	⋆
Space		20	32	62	98	9D	-99	⋆	⋆	⋆	⋆	⋆
#	⋆	23	35	5A	90	A5	-91				⋆	⋆
*	⋆	2A	42	5B	91	A4	-92				⋆	⋆
+	⋆	2B	43	3A	58	C5	-59				⋆	⋆
,		2C	44	66	102	99	-103	⋆	⋆	⋆	⋆	⋆
,	⋆	2C	44	5C	92	A3	-93				⋆	⋆
-		2D	45	17	23	E8	-24	⋆	⋆	⋆	⋆	⋆
-	⋆	2D	45	3B	59	C4	-60				⋆	⋆
.		2E	46	67	103	98	-104	⋆	⋆	⋆	⋆	⋆
.	⋆	2E	46	4C	76	B3	-77				⋆	⋆
/		2F	47	68	104	97	-105	⋆	⋆	⋆	⋆	⋆
/	⋆	2F	47	4A	74	B5	-75				⋆	⋆
0		30	48	27	39	D8	-40	⋆	⋆		⋆	⋆
0		30	48	27	39	D8	-40			⋆		
0	⋆	30	48	6A	106	95	-107				⋆	⋆
1		31	49	30	48	CF	-49	⋆	⋆	⋆	⋆	⋆
1	⋆	31	49	6B	107	94	-108				⋆	⋆
2		32	50	31	49	CE	-50	⋆	⋆	⋆	⋆	⋆
2	⋆	32	50	7C	124	83	-125				⋆	⋆
3		33	51	11	17	EE	-18	⋆	⋆	⋆	⋆	⋆
3	⋆	33	51	6C	108	93	-109				⋆	⋆
4		34	52	12	18	ED	-19	⋆	⋆	⋆	⋆	⋆
4	⋆	34	52	7A	122	85	-123				⋆	⋆
5		35	53	13	19	EC	-20	⋆	⋆	⋆	⋆	⋆
5	⋆	35	53	7B	123	84	-124				⋆	⋆
6		36	54	34	52	CB	-53	⋆	⋆	⋆	⋆	⋆
6	⋆	36	54	1A	26	E5	-27				⋆	⋆
7		37	55	24	36	DB	-37	⋆	⋆	⋆	⋆	⋆
7	⋆	37	55	1B	27	E4	-28				⋆	⋆
8		38	56	15	21	EA	-22	⋆	⋆	⋆	⋆	⋆

Table D.11: BBC Micro Keyboard codes (continued)

		ASCII		Code		INKEY						
Key	Keypad	Hex	Dec	Hex	Dec	Hex	Dec	BBC B	BBC B+	Electron	Master	Compact
8	⋆	38	56	2A	42	D5	-43				⋆	⋆
9		39	57	26	38	D9	-39	⋆	⋆	⋆	⋆	⋆
9	⋆	39	57	2B	43	D4	-44				⋆	⋆
:		3A	58	48	72	B7	-73	⋆	⋆	⋆	⋆	⋆
;		3B	59	57	87	A8	-88	⋆	⋆	⋆	⋆	⋆
@		40	64	47	71	B8	-72	⋆	⋆		⋆	⋆
A		41	65	41	65	BE	-66	⋆	⋆	⋆	⋆	⋆
B		42	66	64	100	9B	-101	⋆	⋆	⋆	⋆	⋆
C		43	67	52	82	AD	-83	⋆	⋆	⋆	⋆	⋆
D		44	68	32	50	CD	-51	⋆	⋆	⋆	⋆	⋆
E		45	69	22	34	DD	-35	⋆	⋆	⋆	⋆	⋆
F		46	70	43	67	BC	-68	⋆	⋆	⋆	⋆	⋆
G		47	71	53	83	AC	-84	⋆	⋆	⋆	⋆	⋆
H		48	72	54	84	AB	-85	⋆	⋆	⋆	⋆	⋆
I		49	73	25	37	DA	-38	⋆	⋆	⋆	⋆	⋆
J		4A	74	45	69	BA	-70	⋆	⋆	⋆	⋆	⋆
K		4B	75	46	70	B9	-71	⋆	⋆	⋆	⋆	⋆
L		4C	76	56	86	A9	-87	⋆	⋆	⋆	⋆	⋆
M		4D	77	65	101	9A	-102	⋆	⋆	⋆	⋆	⋆
N		4E	78	55	85	AA	-86	⋆	⋆	⋆	⋆	⋆
O		4F	79	36	54	C9	-55	⋆	⋆	⋆	⋆	⋆
P		50	80	37	55	C8	-56	⋆	⋆	⋆	⋆	⋆
Q		51	81	10	16	EF	-17	⋆	⋆	⋆	⋆	⋆
R		52	82	33	51	CC	-52	⋆	⋆	⋆	⋆	⋆
S		53	83	51	81	AE	-82	⋆	⋆	⋆	⋆	⋆
T		54	84	23	35	DC	-36	⋆	⋆	⋆	⋆	⋆
U		55	85	35	53	CA	-54	⋆	⋆	⋆	⋆	⋆
V		56	86	63	99	9C	-100	⋆	⋆	⋆	⋆	⋆
W		57	87	21	33	DE	-34	⋆	⋆	⋆	⋆	⋆
X		58	88	42	66	BD	-67	⋆	⋆	⋆	⋆	⋆
Y		59	89	44	68	BB	-69	⋆	⋆	⋆	⋆	⋆
Z		5A	90	61	97	9E	-98	⋆	⋆	⋆	⋆	⋆
[		5B	91	38	56	C7	-57	⋆	⋆		⋆	⋆
\		5C	92	78	120	87	-121	⋆	⋆		⋆	⋆

Table D.11: BBC Micro Keyboard codes (continued)

		ASCII		Code		INKEY						
Key	Keypad	Hex	Dec	Hex	Dec	Hex	Dec	BBC B	BBC B+	Electron	Master	Compact
]		5D	93	58	88	A7	-89	⋆	⋆		⋆	⋆
^		5E	94	18	24	E7	-25	⋆	⋆		⋆	⋆
_		5F	95	28	40	D7	-41	⋆	⋆		⋆	⋆
Delete		7F	127	59	89	A6	-90	⋆	⋆	⋆	⋆	⋆
Delete	⋆	7F	127	4B	75	B4	-76				⋆	⋆
Copy		87	135	69	105	96	-106	⋆	⋆		⋆	⋆
Copy		87	135	69	105	96	-106			⋆		
Curs Left		88	136	19	25	E6	-26	⋆	⋆		⋆	⋆
Curs Left		88	136	19	25	E6	-26			⋆		
Curs Right		89	137	79	121	86	-122	⋆	⋆		⋆	⋆
Curs Right		89	137	79	121	86	-122			⋆		
Curs Down		8A	138	29	41	D6	-42	⋆	⋆		⋆	⋆
Curs Down		8A	138	29	41	D6	-42			⋆		
Curs Up		8B	139	39	57	C6	-58	⋆	⋆			
Curs Up		8B	139	39	57	C6	-58				⋆	⋆
Curs Up		8B	139	39	57	C6	-58			⋆		
f0				20	32	DF	-33	⋆	⋆		⋆	⋆
f1				71	113	8E	-114	⋆	⋆		⋆	⋆
f2				72	114	8D	-115	⋆	⋆		⋆	⋆
f3				73	115	8C	-116	⋆	⋆		⋆	⋆
f4				14	20	EB	-21	⋆	⋆		⋆	⋆
f5				74	116	8B	-117	⋆	⋆		⋆	⋆
f6				75	117	8A	-118	⋆	⋆		⋆	⋆
f7				16	22	E9	-23	⋆	⋆		⋆	⋆
f8				76	118	89	-119	⋆	⋆		⋆	⋆
f9				77	119	88	-120	⋆	⋆		⋆	⋆
Caps Lock				40	64	BF	-65	⋆	⋆		⋆	⋆
CapsLk Func				40	64	BF	-65			⋆		
Control				01	1	FE	-2	⋆	⋆	⋆	⋆	⋆
Shift				00	0	FF	-1	⋆	⋆	⋆	⋆	⋆
Shift Lock				50	80	AF	-81	⋆	⋆		⋆	⋆

D.8 VDU code summary

Dec	Hex	Ctrl	ASCII	+bytes	Meaning
0	0	@	NULL	0	Does nothing
1	1	A	SOH	1	Send next character to printer only
2	2	B	STX	0	Enable printer
3	3	C	ETX	0	Disable printer
4	4	D	EOT	0	Write text at text cursor
5	5	E	ENQ	0	Write text at graphics cursor
6	6	F	ACK	0	Enable VDU Driver
7	7	G	BEL	0	Make a short beep
8	8	H	BS	0	Backspace cursor one character
9	9	I	HT	0	Forward cursor one character
10	A	J	LF	0	Move cursor down one line
11	B	K	VT	0	Move cursor up one line
12	C	L	FF	0	Clear text area
13	D	M	CR	0	Move cursor to start of line
14	E	N	SO	0	Page mode on
15	F	O	SO	0	Page mode off
16	10	P	DLE	0	Clear graphics area
17	11	Q	DC1	1	Define text colour
18	12	R	DC2	2	Define graphics colour
19	13	S	DC3	5	Define logical colour
20	14	T	DC4	0	Restore default logical colours
21	15	U	NAK	0	Disable VDU driver or delete curr. line
22	16	V	SYN	1	Select screen mode
23	17	W	ETB	9	Reprogram display character
24	18	X	CAN	8	Define graphics window
25	19	Y	EM	5	PLOT K,x,y
26	1A	Z	SUB	0	Restore default windows
27	1B	[	ESC	0	Does nothing
28	1C	\	FS	4	Define text window
29	1D	]	GS	4	Define graphics window
30	1E	^	RS	0	Home text cursor to top left
31	1F	_	US	2	Move text cursor to x,y
127	7F		DEL	0	Backspace and delete

+bytes – number of additional bytes needed to complete the VDU code

D.8.1 VDU 23

As well as define the character definitions, VDU 23 can perform additional functions.

Set cursor state

To set the cursor state to c:

```
VDU 23,1,c,0;0;0;0
```

Table D.12: VDU 23,1 cursor states

c	State	
0	Hide cursor	
1	Display cursor	
2	Display steady cursor	Electron only
3	Display flashing cursor	

Set 6845 register

On the BBC Microcomputer, VDU 23 can also be used to access the 6845 CRTC chip. To set register *r* to value *v*:

```
VDU 23,0,r,v;0;0;0
```

D.8.2 VDU Erratta

The BBC Model B User Guide[10] Appendix L states VDU 19 expects 3 extra bytes, whilst the New Advanced User Guide[4] shows 5.

It also shows VDU 20 expects 4 extra bytes when this call actually expects none.

[10] *BBC Micro. User Guide*, (Oct. 1984) P.501

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022)

D.9 BBC Master 128 CMOS RAM BBC Master Compact EEPROM format

Table D.13 shows the content of the settings stored in the CMOS RAM of the BBC Master 128 and the EEPROM of the BBC Master Compact.

Paged ROM

Addresses `0x06` and `0x07` indicate which Paged ROMs are plugged in or marked as offline. When a bit is set then the rom is "inserted" and is usable.

Address `0x06` deals with ROMs 0 ... 7 and `0x07` ROMs 8 ... 15.

Reserved entries

Addresses `0x14` ...`0x1D` are reserved for new filesystems whilst `0x2E` ...`0x31` are for user applications.

BBC Master Compact

The BBC Master Compact uses an EEPROM which has more addresses than the BBC Master 128 but those addresses are not documented anywhere – presumably they are not used by MOS.

Table D.13: BBC Master CMOS Ram

Dec	Hex	Bits	Value	Function
0	00			Econet station number
1	01			File server station number
2	02			File server network number
3	03			Printer server station number
4	04			Printer server network number
5	05	0 ... 3		Default filing system ROM
5	05	4 ... 7		Default language system ROM
6	06			ROM 0 ... 7 inserted

Table D.13: BBC Master CMOS Ram (continued)

Dec	Hex	Bits	Value	Function
7	07			ROM 8 ... 15 inserted
8	08			Allocated to EDIT ROM
9	09			Reserved for telecom apps
10	0A	0 ... 2	0 ... 7	Default screen mode
10	0A	3	0	Main screen
10	0A	3	1	Shadow screen
10	0A	4	0	TV Interlace on
10	0A	5 ... 7		*TV 100=-4 to 011=+3
11	0B	0 ... 2		FDrive settings
11	0B	3 ... 5	001	Shift Caps
11	0B	3 ... 5	010	No lock
11	0B	3 ... 5	100	Caps Lock
11	0B	6	0	ADFS load dir at start
11	0B	6	1	ADFS no dir at start
11	0B	7	0	Default to hard drive
11	0B	7	1	Default to floppy drive
12	0C			Keyboard auto-repeat delay
13	0D			Keyboard auto-repeat rate
14	0E			Printer ignore character if $0x0F_1$ set
15	0F	0	0	No Tube
15	0F	0	1	Tube active
15	0F	1	0	Use printer ignore char
15	0F	1	1	Do not use printer ignore char
15	0F	2 ... 4		Default serial baud rate
15	0F	5 ... 7		Default serial baud rate
16	10	0		Unused
16	10	1	0	BEEP quiet
16	10	1	1	BEEP loud
16	10	2	0	Internal Tube
16	10	2	1	External Tube
16	10	3	0	Scrolling enabled
16	10	3	1	Scrolling disabled
16	10	4	0	No boot on power up
16	10	4	1	Boot on power up
16	10	5 ... 7		Default serial data format
17	11			ANFS Settings
17	11	0	0	No workspace

Table D.13: BBC Master CMOS Ram (continued)

Dec	Hex	Bits	Value	Function
17	11	0	1	2 page workspace
17	11	1	1	Run FINDLIB
17	11	2	0	Workspace 0xB/C
17	11	2	1	Workspace 0xE/F
17	11	3 ... 5	1	Unused
17	11	6	0	UnProtected
17	11	6	1	Protected
17	11	7	1	Display version
18	12			Master Compact Joystick settings
18	12	0 ... 3		Joystick Movement Speed
18	12	4		Unused
18	12	5	0	Proportional emulation
18	12	5	1	Switched mode
18	12	6 ... 7		Reserved
19	13			Reserved
20	14			Reserved for new filesystems
30	1E			Allocated to ROM 0
31	1F			Allocated to ROM 1
32	20			Allocated to ROM 2
33	21			Allocated to ROM 3
34	22			Allocated to ROM 4
35	23			Allocated to ROM 5
36	24			Allocated to ROM 6
37	25			Allocated to ROM 7
38	26			Allocated to ROM 8
39	27			Allocated to ROM 9
40	28			Allocated to ROM 10
41	29			Allocated to ROM 11
42	2A			Allocated to ROM 12
43	2B			Allocated to ROM 13
44	2C			Allocated to ROM 14
45	2D			Allocated to ROM 15
46	2E			Free for user applications
47	2F			Free for user applications
48	30			Free for user applications
49	31			Free for user applications

Page intentionally blank

Filesystem Commands E

The commands in table Table E.1 on page 998 are commands that relate to the current file system, whilst table Table E.1 on the next page lists the commands that select a specific filesystem.

Table E.3 on the following page describes the values of the system compatibility columns.

E.1 BBC Model B

Any Disc file system requires not just the appropriate ROM but also the disc interface hardware installed inside.

The BBC B primarily used $5\frac{1}{4}$" discs as they were the smallest disc format at that time.

E.2 Acorn Electron

Any Disc file system requires not just the ROM but also a disc interface installed, be it either the Acorn +3 expansion interface or a third party equivalent which usually plugs into a cartridge port on the +1 expansion interface.

The Electron primarily used $3\frac{1}{2}$" discs as they were used with the +3 interface.

E.3 BBC Master Series

The BBC Master Series came with the disc interface hardware as standard.

The Masters used $5\frac{1}{4}$" discs mainly due to most BBC software being on that format. The Master Compact however had a $3\frac{1}{2}$" drive built in.

Table E.1: Common filesystem commands compatibility matrix

Command	Function	BBC B	Electron	Master	Section	Page
*BUILD	Create file from keyboard input	DFS		⋆	35.1.3	792
*CAT	Display filesystem catalog	⋆	⋆	⋆	32.0.1	784
*EXEC	Take input from file	⋆	⋆	⋆	32.0.2	784
*LOAD	Load file into memory	⋆	⋆	⋆	32.0.3	784
*OPT	Set filesystem options	⋆	⋆	⋆	32.0.4	785
*RUN	Execute machine code from file	⋆	⋆	⋆	32.0.5	786
*SAVE	Save memory to file	⋆	⋆	⋆	32.0.6	786
*SPOOL	Write VDU output to file	⋆	⋆	⋆	32.0.7	786

Table E.2: Filesystem selection command compatibility matrix

Command	Function	BBC B	Electron	Master	Section	Page
*ADFS	Select Acorn Advanced DFS	ADFS	+3	⋆	36	811
*DFS	Select Acorn DFS	DFS	DFS	⋆	35	791
*LANMAN	Select LanManager filesystem	*N/A*	*N/A*	SPROW	37	813
*ROM	Select ROM filesystem	⋆	⋆	⋆	**??**	**??**
*TAPE	Select Tape filesystem	⋆	⋆	⋆	**??**	**??**
*TAPE3	run at 300bps	⋆	*N/A*	⋆		
*TAPE12	run at 1200bps	⋆	*N/A*	⋆		

Table E.3: Definitons of compatibility columns

Requirement	Description
⋆	Available on this platform as standard
	Not available as standard, requires expansion ROM
N/A	Not possible due to hardware constraints
ADFS	Requires Acorn Advanced Disk File System (ADFS) ROM
DFS	Requires Acorn Disk File System (DFS) ROM
+1	Acorn Electron +1 Expansion with standard Plus 1 ROM
+3	Acorn Electron +3 Expansion (or equivalent with ADFS ROM)
SPROW	Sprow Ethernet interface for Master Series machines only

BBC Micro and Acorn Electron Ports

F

The following are the pinouts of the various ports on the BBC Micro. They are also the same on the BBC Master[20] and the Acorn Electron.

[20] *Master Series Welcome Guide*, (Mar. 1986) P.239,240

F.1 RGB

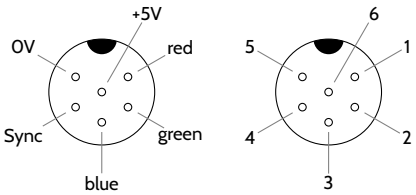


Figure F.1: BBC Micro RGB Port

F.2 RS-423

The BBC Model B and the Master 128 had a RS423 port which provided serial communication capabilities[20].

[20] *Master Series Welcome Guide*, (Mar. 1986)

The Acorn Electron did not have this port as standard, however Pace did produce a cartridge which plugged into the Plus 1 interface and had a plug which had the same pinouts.

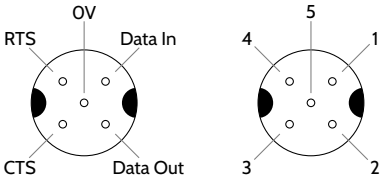


Figure F.2: BBC Micro RS423 Port

F.3 Cassette

The cassette port is common across the entire range and allows for optional control of the tape drive if it has the motor control port available - which many cassette machines had at the time.

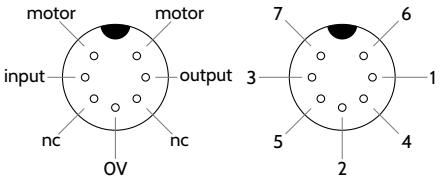


Figure F.3: BBC Micro Cassette Port

The port layout is standardised to the cassette cabling at the time so it was compatible with DIN3 and DIN5 cabling schemes.

These cables however would not allow the computer to control the cassette motor as those pins are not connected.

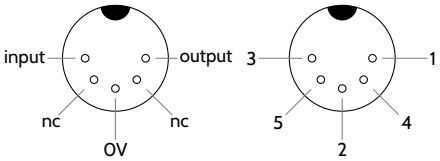


Figure F.4: DIN5 cables compatible with BBC Cassette Port

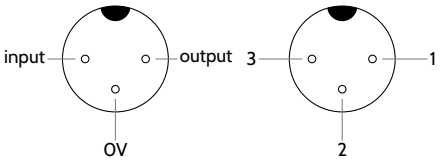


Figure F.5: DIN3 cables compatible with BBC Cassette Port

F.4 Analogue Input

The BBC Micro had an analogue input port which was used for either joysticks, paddles or rudimentary analogue interfacing with devices. The Acorn Electron has the same port as part of the Plus1 expansion.

The input range is between $0V$ ... $1.8V$. When $0V$ then the result of the analogue to digital converter is 0, but at $1.8V$ the ADC returns 65520[1].

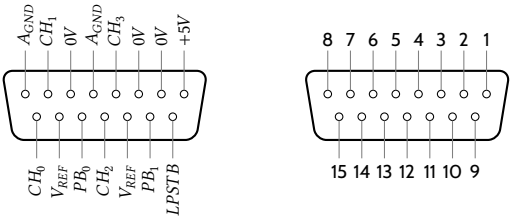


Figure F.6: BBC Micro Analogue Port

1: The BBC ADC returns 0 ... 4095 but to allow for higher-precision ADC's they shifted the result by 4 bits allowing for 16 times the resolution

F.5 Econet

The Econet was Acorn's attempt at networking, supporting up to 254 machines on a network.



Figure F.7: BBC Micro Econet Port

F.5.1 SPROW Ethernet

[18] *Sprow Master 10/100 Net User Guide*, (2008)

A third party interface is the Sprow ethernet[18] card which connects internally in the Master 128 where the Econet interface would.

It replaces this with a custom cable which connects to the original Econet port. Figure Figure F.8 on page 1002 shows the pinouts of the modified port which connects to a standard ethernet cable.



Figure F.8: BBC Micro Econet Port wired with the SPROW ethernet interface

F.6 Acorn Electron Cartridge Slot

The Master cartridge is very similar but is shown separately as there are some differences with the slots.

The Plus 1 expansion for the Acorn Electron introduced cartridge slots to the Acorn range of microcomputers. The BBC Master 128 then used the same cartridges with a few pin changes. However, in most cases, other than specific hardware cartridges, many cartridges for the Electron would also work for the Master,

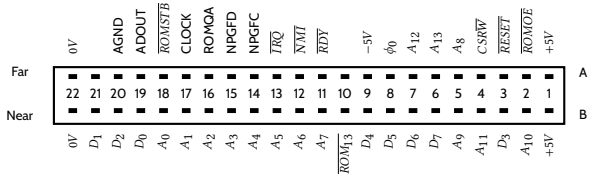

0V AGND ADOUT ROMSTB CLOCK ROMQA NPGFD NPGFC IRQ NMI RDY −5V φ0 A12 A13 A8 CSRW RESET ROMOE +5V
Far
22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
A
Near
B
0V D1 D2 D0 A0 A1 A2 A3 A4 A5 A6 A7 ROM13 D4 D5 D6 D7 A9 A11 D3 A10 +5V

Figure F.9: Electron Master Cartridge Slot

Table F.1: Acorn Electron Cartridge Slot

Line	Description
$+5V$	Power supply
$-5V$	See Appendix F.9.2 on page 1006
$D_{0...7}$	Data Bus
$A_{0...15}$	Address bus
ϕ_0	Input from ϕ_0 clock
$\overline{RESET}$	System reset if pulled low
$\overline{ROMOE}$	See Appendix F.9.6 on page 1007
$ROMQA$	See Appendix F.9.7 on page 1007
$CSR\overline{W}$	See Appendix F.9.8 on page 1008
$\overline{RDY}$	See Appendix F.9.10 on page 1008
$\overline{NMI}$	Non Maskable Interrupt
$\overline{IRQ}$	Maskable Interrupt
$NPGFC$	See Appendix F.9.11 on page 1008
$NPGFD$	See Appendix F.9.11 on page 1008
$CLOCK$	See Appendix F.9.12 on page 1009
$\overline{ROMSTB}$	Select ROM 13
$ADOUT$	Audio output – See Appendix F.9.16 on page 1011
$AGND$	Audio Ground

F.7 Master 128 Cartridge Slot

The Electron cartridge is very similar but is shown separately as there are some differences with the slots.

The BBC Master 128 has two cartridge slots on the machine, above the numeric keypad.

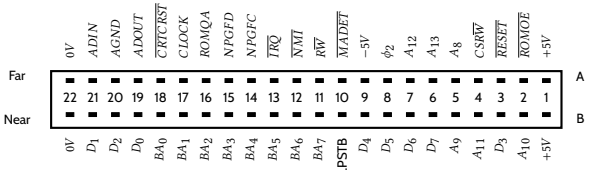

0V ADIN AGND ADOUT CRTCRST CLOCK ROMQA NPGFD NPGFC IRQ NMI RW MADET −5V φ2 A12 A13 A8 CSRW RESET ROMOE +5V
Far
22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
A
Near
B
0V D1 D2 D0 BA0 BA1 BA2 BA3 BA4 BA5 BA6 BA7 LPSTB D4 D5 D6 D7 A9 A11 D3 A10 +5V

Figure F.10: BBC Master Cartridge Slot

Table F.2: BBC Master Cartridge Slot

Line	Description
$+5V$	Power supply
$-5V$	See Appendix F.9.2 on page 1006
$D_{0...7}$	Data Bus
$BA_{0...7}$ $A_{8...15}$	Address bus
ϕ_2	Input from ϕ_2 clock
$\overline{RESET}$	System reset if pulled low
$\overline{ROMOE}$	See Appendix F.9.6 on page 1007
$ROMQA$	See Appendix F.9.7 on page 1007
$CSR\overline{W}$	See Appendix F.9.8 on page 1008
$\overline{RW}$	See Appendix F.9.9 on page 1008
$\overline{NMI}$	Non Maskable Interrupt
$\overline{IRQ}$	Maskable Interrupt
$NPGFC$	See Appendix F.9.11 on page 1008
$NPGFD$	See Appendix F.9.11 on page 1008
$CLOCK$	See Appendix F.9.12 on page 1009
$\overline{CRTCRST}$	System CRTC reset for genlock use
$\overline{MADET}$	See Appendix F.9.13 on page 1010
$ADOUT$	Audio output – See Appendix F.9.16 on page 1011
$AGND$	Audio Ground
$ADIN$	Audio In
$LPSTB$	See Appendix F.9.14 on page 1010

F.8 Master Compact Cartridge Slot

The Master Compact's cartridge slot is wider than the one used on the Electron and Master so are not directly compatible.

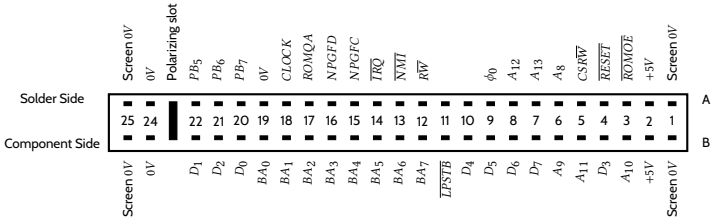


Figure F.11: Master Compact Cartridge Slot

Table F.3: Master Compact Cartridge Slot

Line	Description
$+5V$	Power supply
$D_{0...7}$	Data Bus
$A_{0...15}$	Address bus
ϕ_0	Input from ϕ_0 clock
$\overline{RESET}$	System reset if pulled low
$\overline{ROMOE}$	See Appendix F.9.6 on page 1007
$ROMQA$	See Appendix F.9.7 on page 1007
$CSR\overline{W}$	See Appendix F.9.8 on page 1008
$\overline{RDY}$	See Appendix F.9.10 on page 1008
$\overline{NMI}$	Non Maskable Interrupt
$\overline{IRQ}$	Maskable Interrupt
$NPGFC$	See Appendix F.9.11 on page 1008
$NPGFD$	See Appendix F.9.11 on page 1008
$CLOCK$	See Appendix F.9.12 on page 1009
$\overline{ROMSTB}$	Select ROM 13
$ADOUT$	Audio output – See Appendix F.9.16 on page 1011
$AGND$	Audio Ground

F.9 Electron, Master and Compact cartridge slot notes

These notes[4] apply to the cartridge slots on the Acorn Electron[2], BBC Master[3], and the BBC Master Compact[4].

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) P.416-418

2: Acorn Electron
Appendix F.6 on page 1003

3: BBC Master
Appendix F.7 on page 1004

4: BBC Master Compact
Appendix F.8 on page 1005

F.9.1 $+5V$ supply

The $+5V$ is connected to the system $+5V$ supply. The max load that can be drawn differs on which system is being used. Table F.4 shows the max load for each system.

Table F.4: $+5V$ Max Load

System	Max Load
Electron	$50mA$
Master	$150mA$
Compact	$200mA$

F.9.2 $-5V$ supply

The $-5V$ supply may not be present on all Acorn cartridge interfaces, so it's been suggested to generate it from $+5V$ with a small inverter. It is not connected on the Master Compact.

If it is used, then a maximum of $20mA$ can be drawn from this pin.

F.9.3 Address bus

On the BBC Master, Bits $BA_{0...7}$ of the Address Bus is buffered whilst bits $A_{8...15}$ are not. The buffered bits are guaranteed to hold addresses for $125ns$ after ϕ_2 goes low.

On the Electron the entire address bus $A_{0...15}$ is unbuffered.

F.9.4 Data bus

The Data Bus lines $D_{0...7}$ are all TTL level.

F.9.5 phi clocks

The ϕ clock comes from the processors clock.

On the BBC Master and Master Compact the clock is a CMOS level signal from the ϕ_2 clock.

On the Electron it is from the ϕ_0 clock.

Table F.5: Cartridge ϕ clock signal source

System	Source
Electron	ϕ_0
Master	ϕ_2
Compact	ϕ_2

F.9.6 $\overline{ROMOE}$

Output enable used to switch on the output buffers of cartridge memory devices when low during the ϕ_2 period of the system clock[5].

This can be used directly as a chip enable to 32K EPROM's whilst using the *ROMQA* pin to select which half of the EPROM is being accessed.

5: Not guaranteed low at other times

F.9.7 *ROMQA*

This is a memory paging select bit.

It is the least significant bit of the ROM select latch. As each cartridge has two Paged ROM slots allocated to it, this bit selects which rom is to be active.

Along with the $\overline{ROMOE}$ line, this can be used to enable 32K EPROM's to be used instead of 16K EPROM's. In this instance, the EPROM is enabled when $\overline{ROMOE}$ is pulled low but *ROMQA* is connected to the EPROM's most significant address line. Then when *ROMQA* is 0 the lower half of the EPROM is used and when 1 the upper half.

Section 23.7.1 in The New Advanced User Guide[4] covers this as well as using 128K EPROM's in cartridges.

Table F.6: ROM select latch address

System	Address
Master	`0xFE30`
Electron	`0xFE05`

[4] *Adv. User Guide, BBC Master, Comp., B, B+, Electron*, Dickens (Adder Publishing Limited June 2022) 23.7.1 P.414

F.9.8 $CSR\overline{W}$

On the Electron this is the 6502's $R/\overline{W}$ line.

On the Master this varies on the memory being used. For accesses to `0xFC00` ...`0xFEFF` this is equivalent to the CPU's $R/\overline{W}$ line. When accessing all other areas of memory it is an active high chip select for memory devices during ϕ_2, but is not guaranteed low at other times.

F.9.9 $R\overline{W}$

On the Master and Compact this is the Read/Write data direction control input. Cartridges are being written to when this is low, and they may drive the bus during ϕ_2 if this signal is high and the cartridge is selected.

F.9.10 $\overline{RDY}$



This shares the same pin as $R\overline{W}$ on the Master

On the Electron, this is an open collector active low to the CPU's WAIT line.

For CMOS CPU's, when it is low this will cause the CPU to extend its cycle.

For NMOS CPU's this will only extend the read cycles.

F.9.11 *NPGFC* and *NPGFD*

These signals are triggered when either page `0xFC` or `0xFD` are selected.

On the Master they only becomes active when bit IFJ in *ACCCON* register is set.

F.9.12 *CLOCK*

The CLOCK pin provides a fixed clock from the system.

For the Electron this is a $16MHz$ clock, whilst on the Compact it's a $8MHz$ clock.

Table F.7: *CLOCK* frequency

System	Frequency
Electron	$16MHz$
Compact	$8MHz$
Master	*See below*

Master CLOCK

On the Master, in normal use this is the same as the Compact, and it is a $8Mhz$ clock sent to the cartridge.

However, if links LK6(0) and LK6(1) are altered then the output differs[9].

[9] *Master Series Ref. Manual Part 1*, (Jan. 1986) P.F.5-2

A–B Linked

$16MHz$ reference by onboard circuitry. This is the default setup.

B–D linked

$16MHz$ clock must be provided on the CLOCK pin A_{17} on the cartridge connectors.

C–D linked

$8MHz$ clock sent to cartridges which is synchronous with the $2MHz\phi_2$ signal that is provided to them on the ϕ line. Link A–B must also be fitted.

F.9.13 $\overline{MADET}$

This pin allows cartridges to detect which machine they are plugged in on.

On the Compact this is always $0V$.

This is not connected on the Electron so will be floating high on that machine.

On the Master, Link 12 determines this line's action. The default for that link is to pull the line low like the Compact. However, if the link is changed to position B then the composite sync signal, at TTL levels, is made available for genlock use.

F.9.14 *LPSTB*

On the Compact this provides the *LPSTB* signal which is the light pen strobe input and is pulled high by a resistor.

On the Master by default this provides the *LPSTB* signal but if link 21 is removed then it can provide a connection between the two cartridges.

On the Electron this is an active low input for selecting ROM number 13. It is low during the active low portion of the ϕ_2 clock and not guaranteed high at other times.

F.9.15 $PB_{5...7}$ User Port

On the Compact, bits PB_5, PB_6 and PB_7 of the User Port are exposed to the cartridge slot as they are not available on the normal port.

F.9.16 Audio I/O

Three lines provide audio to and from the cartridges.

AGND

This line should be used as the ground line for audio circuitry instead of the system $0V$ to minimize audio noise.

On the Electron this is pulled up to $+5V$ with a $4K7\Omega$ resistor.

ADOUT

This provides an audio output line. No significant load should be taken from this pin.

ADIN

This provides an Audio input into the system. The computers audio amplifier impedance is at least $1K\Omega$ and ony one cartridge should use this line at any time.

On the Master this can be used as the output from a Speech Synthesizer.

The pin is not connected on the Electron.

F.10 MM2114N 4Kb 1024x4 static RAM

MM2114N 4Kb 1024x4 static RAM

Page intentionally blank

G Commodore Ports

The following are the layouts of the various ports on the 8-bit Commodore Machines.

G.1 Serial Bus

The Commodore machines, not just the C64, used a serial bus[19] to connect devices like Floppy Drives and Printers to the machine.

[19] *Commodore 64 Programmers Reference Manual*, () Ch.6 P.362

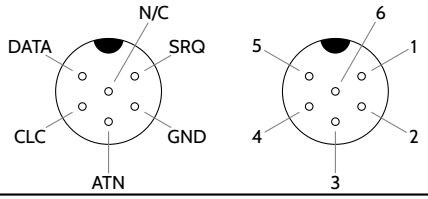
Pin 1 is an input into the bus controller, allowing any device to bring the controller to attention.

Pins 3, 4 and 5 are bidirectional as they are output by the talking device and input to the listening devices on the bus.

Pin 3 is also exposed on the User Port[1].

1: User Port Appendix G.2 on page 1014

Chapter 42 on page 841 describes in detail how the bus operates.



Pin	Label	Direction	Description
1	SRQ	In	Serial Service Request
2	GND		Ground
3	ATN	In / Out	Attention
4	CLK	In / Out	Clock
5	DATA	In / Out	Data
6	–		Not Connected

Figure G.1: Commodore Serial Bus

G.2 User Port

[19] *Commodore 64 Programmers Reference Manual*, () Ch.6 P.360

The Commodore 64 has a user port exposed on the back of the machine[19]. This port can be used to connect the computer to a standard parallel printer, a modem or custom hardware.

Figure G.2 shows the port as viewed from the back of the machine.

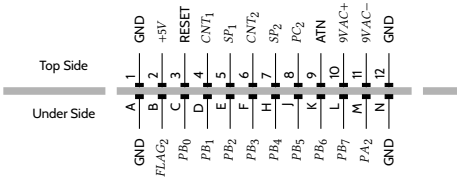


Figure G.2: Commodore User port

Table G.1: C64 User Port Pins

Pin	Line	Description
2	$+5V$	Maximum load of $100mA$
3	RESET	Pull low to reset the computer.
4	CNT_1	Serial port counter from CIA#1
5	SP_1	Serial port from CIA#1
6	CNT_2	Serial port counter from CIA#2
7	SP_2	Serial port from CIA#2
8	PC_2	Handshaking line from CIA#2
9	ATN	The Attention line on the Serial Bus
10	$9VAC$	positive phase, max load of $50mA$
11	$9VAC$	negative phase, max load of $50mA$
B	$FLAG_2$	
C–L	$D_{0...7}$	Port B of CIA#1
M	PA_2	

Pin 9 ATN is described in Serial Bus Appendix G.1.

G.3 RS-232 Port

The Commodore 64 can use its user port as a serial port[19]. This uses only the pins on the underside of the connector.

[19] *Commodore 64 Programmers Reference Manual*, () Ch.6 P.354



Warning

I've called this section RS-232 only to differenciate it from the SerialBus.

These signals are still only at TTL levels so require the correct circuitry to bring it inline with actual RS-232 or RS-423 levels.

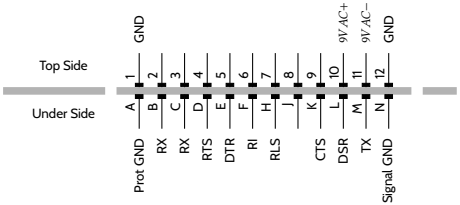


Figure G.3: Commodore RS-232 port

Table G.2: C64 User Port Pins for RS-232

Pin	Line	Description	Dir
A	Prot GND	Protective Ground	
B	RX	Received Data	I
C	RX	Received Data	I
D	RTS	Request to Send	O
E	DTR	Data Terminal Ready	O
F	RI	Ring Indicator	I
H	RLS	Received Line Signal	I
K	CTS	Clear To Send	I
L	DSR	Data Set Ready	I
M	TX	Transmitted Data	O
N	Signal GND	Signal Ground	

The pins on the top side like power, RESET and ATN are still connected, just not used for serial communications

G.4 Expansion Connector

Officially called the Expansion connector, this port exposes the address and data bus of the processor to external hardware. This hardware was usually ROM cartridges, although other expansion hardware exist to connect to this port like RAM expansion[2].

Figure G.4 shows the port as viewed from the back of the machine.

2: I have a 2Mb expansion here, yes not a typo!

Any entry with an overline *e.g.* $\overline{PIN}$ is active low.

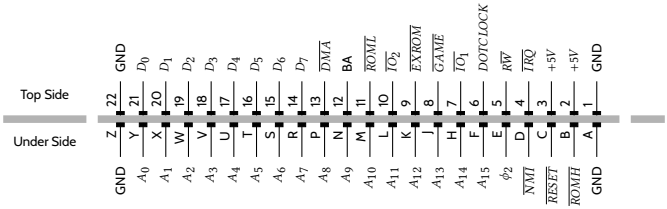


Figure G.4: Commodore 64 Expansion Connector

Table G.3: C64 Expansion Port Pins

Pin	Line	Description	Buffered	Load
1, 22, A, Z	GND	System ground		
2, 3	$+5V$	Total UserPort and Cartridge draw $450mA$		
4	$\overline{IRQ}$	Maskable Interrupt		
5	$R\overline{W}$	High read, low write		
6	DOTCLOCK	$8.18MHz$ video dot clock		
7	IO_1	I/O Block 1 `0xDEnn`	N	
8	$\overline{GAME}$			LS TTL Input
9	$\overline{EXROM}$			LS TTL Input
10	IO_2	I/O Block 2 `0xDFnn`	Y	LS TTL Output
11	$\overline{ROML}$	8k decoded RAM/ROM `0x8000`	Y	LS TTL Output
12	BA	Bus Available signal from VIC-II	N	1 LS Load Max
13	$\overline{DMA}$	Direct Memory Access request		LS TTL Input
14…21	$D_{0...7}$	Data Bus	N	1 LS TTL Load Max
B	$\overline{ROMH}$	8k decoded RAM/ROM `0xE000`	Y	
C	$\overline{RESET}$	6502 Reset pin	X	See below
D	$\overline{NMI}$	6502 Non-Maskable Interrupt		
E	ϕ_2	6502 Phase 2 system clock		
F…Y	$A_{0...15}$	Address bus		

G.4.1 BA Bus Available

The BA (Bus Available) signal from the VIC-II chip goes low 3 cycles before the VIC-II takes over the system busses and remains low until the VIC-II has finished fetching display information.

G.4.2 DMA Direct Memory Access

If $\overline{DMA}$ is pulled low the address bus, data bus and the Read/Write line of the 6510 processor enters high-impedance state mode to allow an external processor to take control of the system busses.

This line should only be pulled low when the ϕ_2 clock is low. Also, since the VIC-II chip will continue to perform display DMS, the external device must conform to the VIC-II timing.

G.4.3 Notes

According to the reference manual[19] $\overline{RESET}$ and $\overline{NMI}$ states they are buffered for output but unbuffered input. I'm not sure how that works with buffering in one direction only? Also they are input into the processor.

[19] *Commodore 64 Programmers Reference Manual*, () Ch.6 P.367

The $\overline{IRQ}$ line doesn't say anthing about buffering, even though it's similar to $\overline{NMI}$ in use.

Also some entries on that same page say "1 ls ttl" - I'm presuming that is LS as in the LS series of TTL chips? The scan copy of that manual I have is not that great.

Page intentionally blank

Sinclair ZX-81 Ports

H

The following are the layouts of the various ports on the Sinclair ZX-81 Machines.

H.1 Edge Connector

This is the edge connector[21] for the ZX-81.

[21] *ZX81 Basic Programming*, Vickers (Sinclair Research Limited 1981) P.167

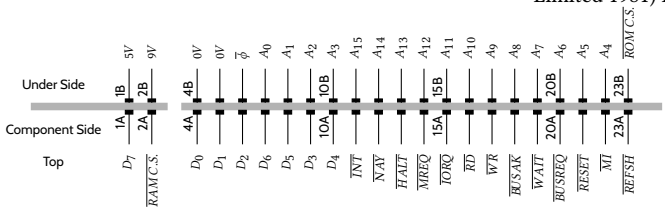


Figure H.1: ZX-81 Edge Connector

Page intentionally blank

See Chapter 43 on page 845 for a full description on how the keyboard works.

Table I.1: Spectrum Keyboard codes

	ASCII		BASIC		Shifted		Hardware	
Character	Hex	Dec	Command	Symbol	Upper	Lower	Port	Bit
ENTER	0D	13					BFFE	0
SPACE	20	32					7FFE	0
0	30	48		_	DELETE	FORMAT	EFFE	0
1	31	49		!	EDIT	DEF FN	F7FE	0
2	32	50		@	CAPS LOCK	FN	F7FE	1
3	33	51		#	TRUE VIDEO	LINE	F7FE	2
4	34	52		$	INV. VIDEO	OPEN #	F7FE	3
5	35	53		%	←	CLOSE #	F7FE	4
6	36	54		&	↓	MOVE	EFFE	4
7	37	55		'	↑	ERASE	EFFE	3
8	38	56		(	→	POINT	EFFE	2
9	39	57		)	GRAPHICS	CAT	EFFE	1
A	41	65	NEW	STOP	READ	~	FDFE	0
B	42	66	BORDER	*	BIN	BRIGHT	7FFE	4
C	43	67	CONT	?	LPRINT	PAPER	FEFE	3
D	44	68	DIM	STEP	DATA	\	FDFE	2
E	45	69	REM	>=	TAN	ATN	FBFE	2
F	46	70	FOR	TO	SGN	{	FDFE	3
G	47	71	GOTO	THEN	ABS	}	FDFE	4

Table I.1: Spectrum Keyboard codes (continued)

	ASCII		BASIC		Shifted		Hardware	
Character	Hex	Dec	Command	Symbol	Upper	Lower	Port	Bit
H	48	72	GOSUB	↑	SQR	CIRCLE	BFFE	4
I	49	73	INPUT	AT	CODE	IN	DFFE	2
J	4A	74	LOAD	-	VAL	VAL$	BFFE	3
K	4B	75	LIST	+	LEN	SCREEN$	BFFE	2
L	4C	76	LET	=	USR	ATTR	BFFE	1
M	4D	77	PAUSE	.	PI	INVERSE	7FFE	2
N	4E	78	NEXT	,	INKEY$	OVER	7FFE	3
O	4F	79	POKE	;	PEEK	OUT	DFFE	1
P	50	80	PRINT	"	TAB	©	DFFE	0
Q	51	81	PLOT	<=	SIN	ASN	FBFE	0
R	52	82	RUN	<	INT	VERIFY	FBFE	3
S	53	83	SAVE	NOT	RESTR	|	FDFE	1
T	54	84	RAND	>	RND	MERGE	FBFE	4
U	55	85	IF	OR	CHR$	]	DFFE	3
V	56	86	CLS	/	LLIST	FLASH	FEFE	4
W	57	87	DRAW	<>	COS	ACS	FBFE	1
X	58	88	CLEAR	£	EXP	INK	FEFE	2
Y	59	89	RETURN	AND	STR$	[	DFFE	4
Z	5A	90	COPY	:	LN	BEEP	FEFE	1
SYMBOL SHIFT							7FFE	1
CAPS SHIFT							FEFE	0

J Sinclair Spectrum Ports

The following are the layouts of the various ports on the Sinclair Spectrum Machines. Some of these ports are shared amongst the various models, one are unique to specific machines.

J.1 Monitor

The Monitor port first appeared on the Spectrum 128K[22] and exposed both a Composite Video signal and a TTL RGB signal.

[22] *Spectrum 128 Rom Dissassembly*, Wilson et al. (Feb. 4, 2019) Rom 0 L.18320

The previous machines only had an RF Television output, although it was possible to get a Composite signal from them with minor hardware modifications.

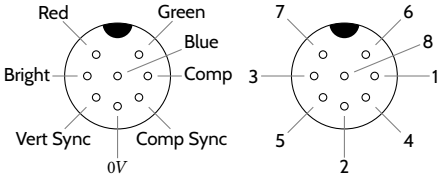


Figure J.1: Spectrum 128K RGB Monitor Port

The Composite output on pin 1 is $1.2V$ peak-peak at 75Ω. The other pins $3 \ldots 8$ are all $5V$ TTL.

J.2 RS-232/Midi

The RS232/MIDI port shown here first appeared on the Spectrum 128K[22] and exposed a serial port which could also handle MIDI.

[22] *Spectrum 128 Rom Dissassembly*, Wilson et al. (Feb. 4, 2019) Rom 0 L.18261

This was not the first RS232 port for the Spectrum, as that appeared for the 48K with the Plus-One interface.

That port used a standard D9 connector, but this used a special connector similar to the UK BT Telephone connectors, except that the locking tab was on the opposite side.

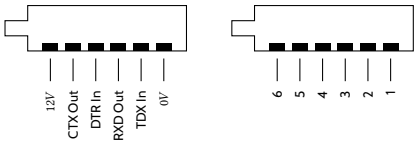


Figure J.2: Spectrum 128K RS232/MIDI Port

J.3 AY-3-8912 Register 14

The RS232 port is controlled by Register 14 on the AY-3-8912 sound generator.

Figure J.3 shows the bits on that register assigned to each bin on the port.

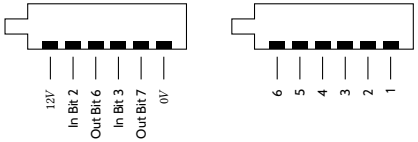


Figure J.3: Spectrum 128K RS232 AY-3-8912 Register 14 Bits

J.4 Keypad

The Spectrum 128K Came with a Keypad/Aux port[22]. For the Keypad this was used to connect to an external numeric keypad.

[22] *Spectrum 128 Rom Dissassembly*, Wilson et al. (Feb. 4, 2019) Rom 0 L.18280

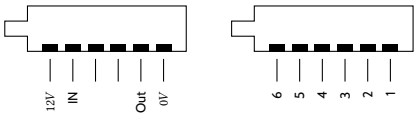


Figure J.4: Spectrum 128K Keypad Port

J.5 AY-3-8912 Register 14

The Keypad port is controlled by Register 14 on the AY-3-8912 sound generator. This enabled the programmer the ability to use the port for their own uses.

Figure J.5 shows the bits on that register assigned to each pin on the port.

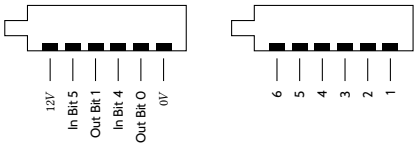


Figure J.5: Spectrum 128K Keypad AY-3-8912 Register 14 Bits

J.6 Amstrad Lightgun

After Amstrad bought Sinclair, the Keypad socket was used to support a light gun[1].

1: I never heard of the light gun until I wrote this.

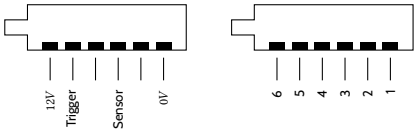


Figure J.6: Spectrum 128K Light Gun

J.7 Spectrum 48K Edge Connector

[23] *ZX Spectrum Basic Programming*, Vickers (Sinclair Research Limited 1983) P160

[22] *Spectrum 128 Rom Dissassembly*, Wilson et al. (Feb. 4, 2019) Rom 0 L.18352

This is the edge connector[23][22] for the Spectrum 48K.

It has some similarities with the ZX-81 connector enabling it to share a few peripherals like the ZX-Printer.

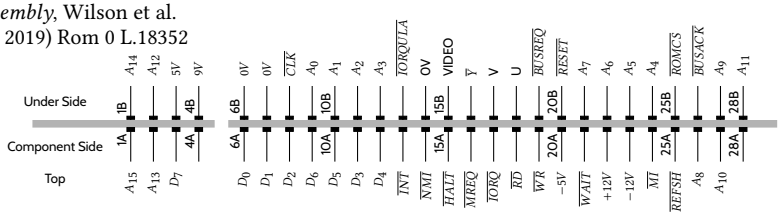


Figure J.7: Spectrum 48K Edge Connector

J.8 Spectrum 128K Edge Connector

[22] *Spectrum 128 Rom Dissassembly*, Wilson et al. (Feb. 4, 2019) Rom 0 L.18352

This is the edge connector[22] for the Spectrum 128K.

It has some similarities with the Spectrum connector enabling it to share many peripherals like the Kempston joystick interface. Some peripherals were not compatible with the 128K like the ZX Interface 1 and the Prism VTX5000 modem which had their own ROMS which were specific to the 48K machine.

Also, the video output from the 48K is absent on this connector.

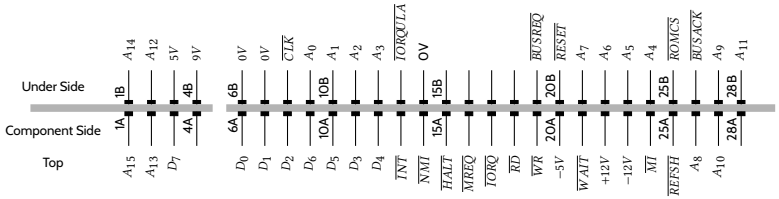


Figure J.8: Spectrum 128K Edge Connector

Text screen planning sheets

K

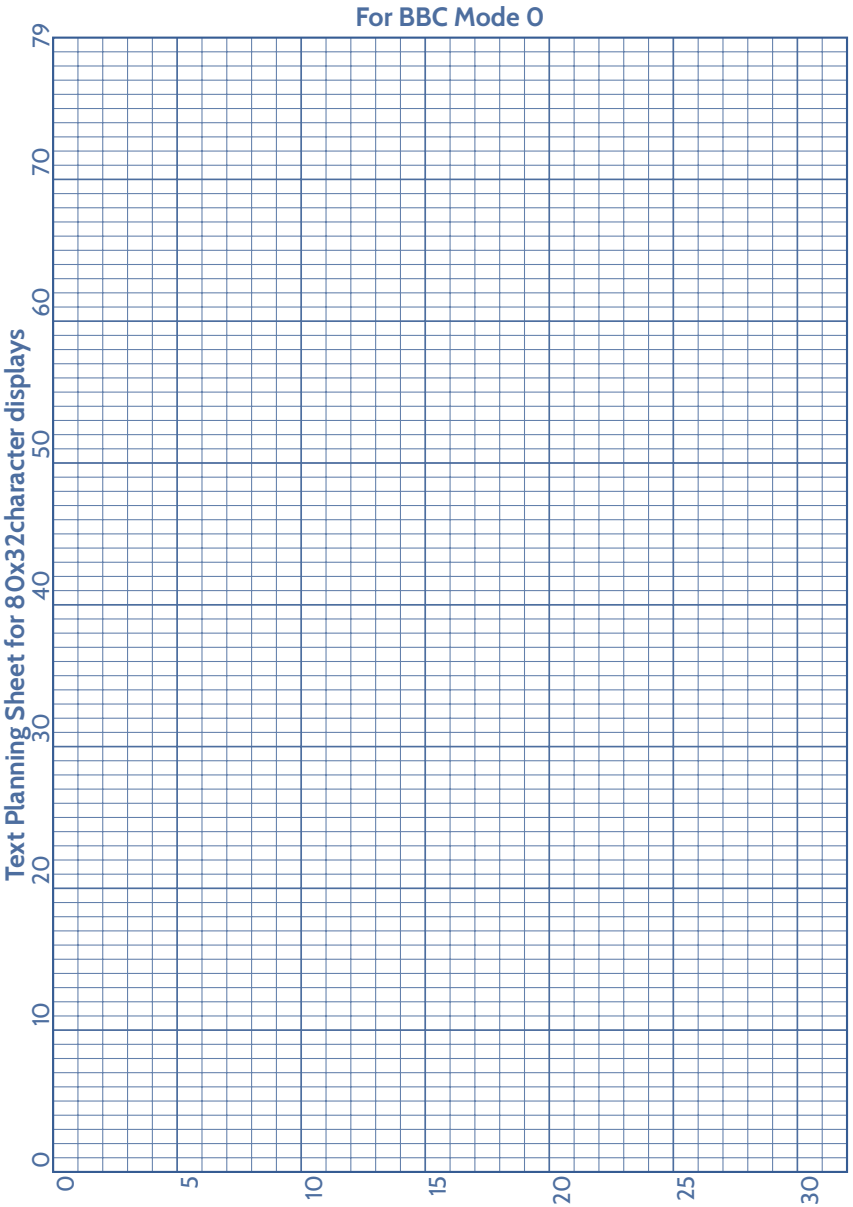
The following set of sheets are based on the planning sheets in the BBC Model B User Guide[10]. Table K.1 on page 1027 lists the screen resolutions for various systems to enable you to select the correct sheet to use.

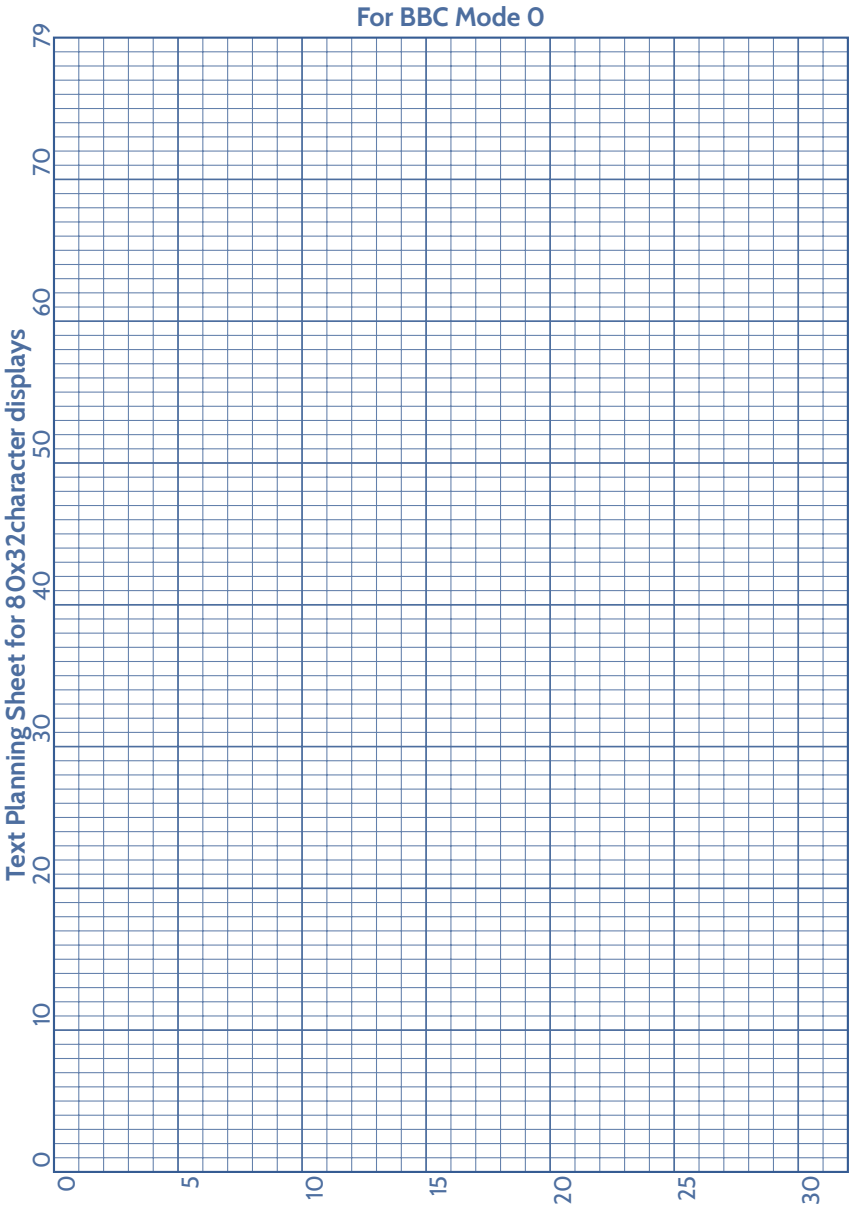
[10] *BBC Micro. User Guide*, (Oct. 1984) Appendix E P.485

Table K.1: Screen Text Resolutions

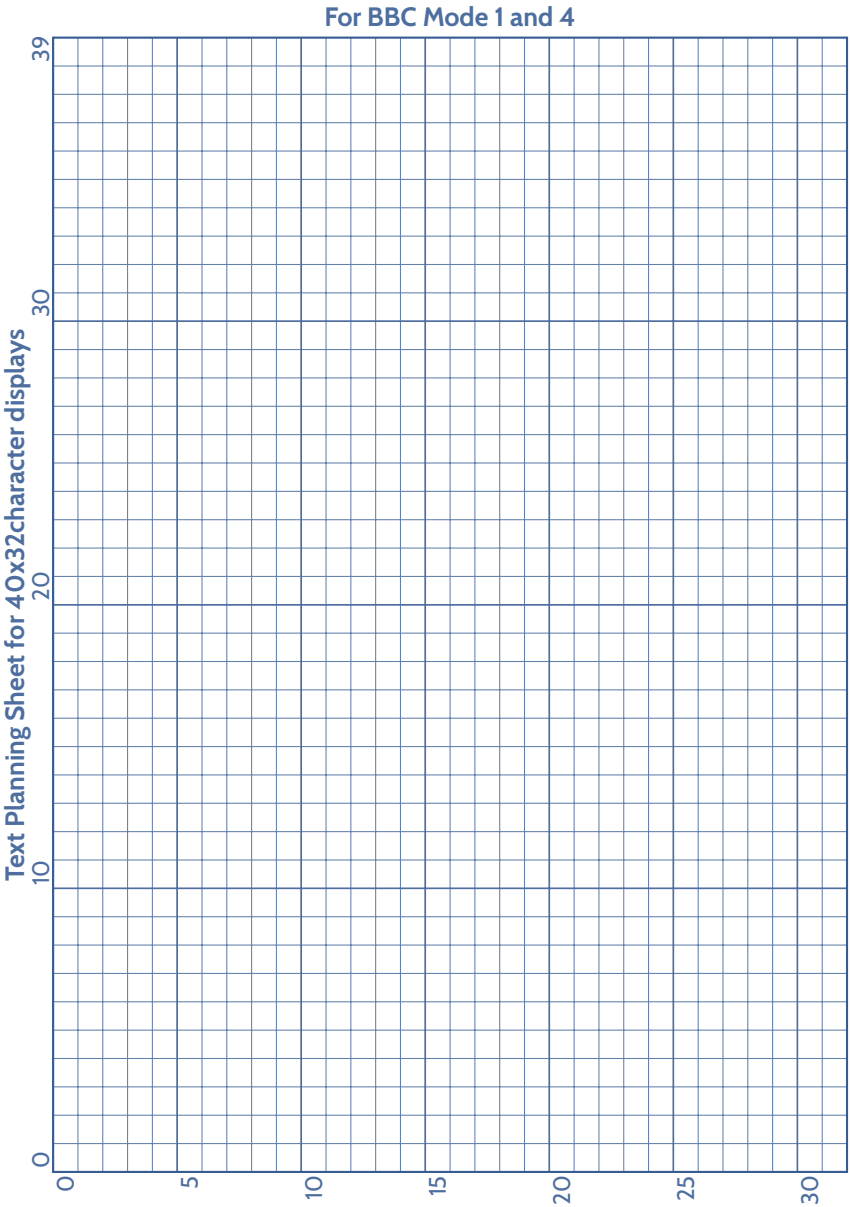
System	Mode	Cols	Rows	Colours
Spectrum		32	24	2
C64	text	40	25	2
BBC Micro	0	80	32	2
BBC Micro	1	80	32	4
BBC Micro	2	20	32	16
BBC Micro	3	80	25	2
BBC Micro	4	40	32	2
BBC Micro	5	20	32	4
BBC Micro	6	40	25	2
BBC Micro	7	40	25	Teletext
TI-99/4A		32	24	2

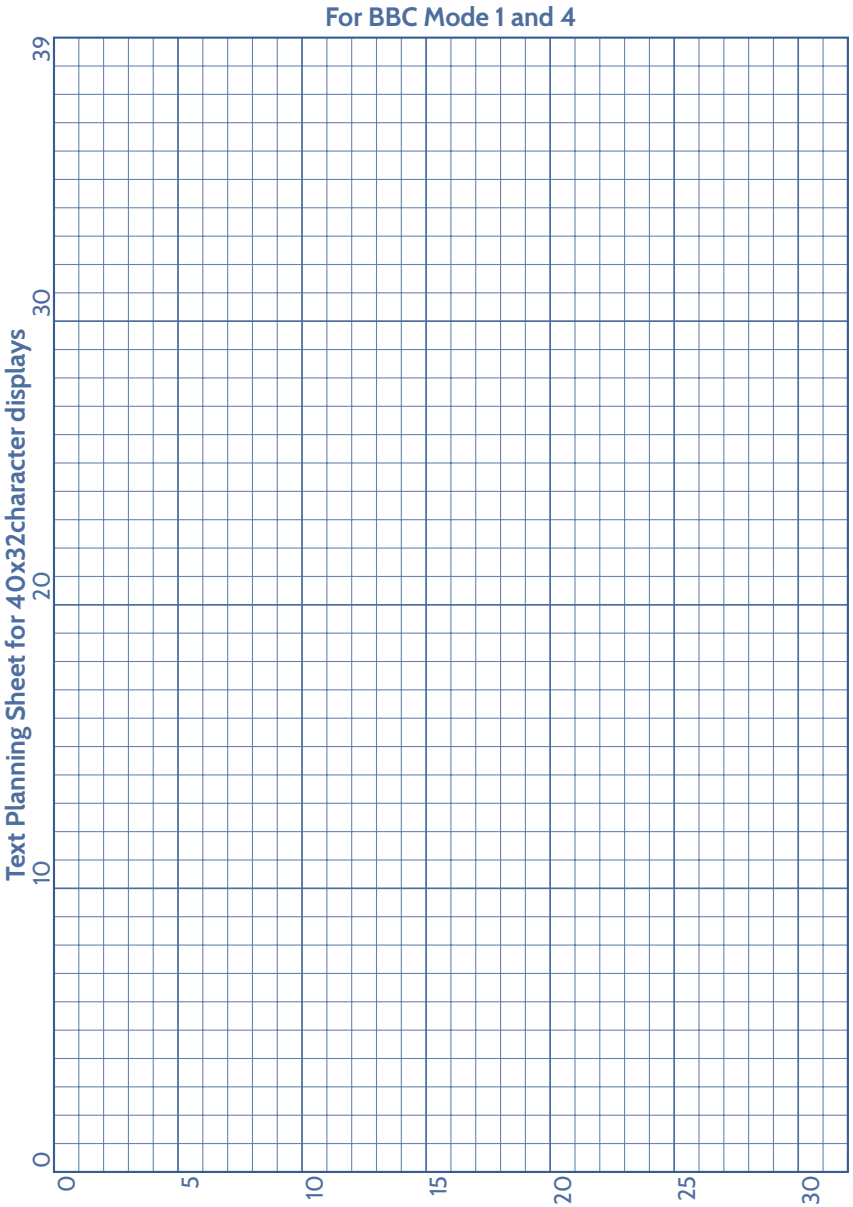
K.1 80x32 text screen planning sheet



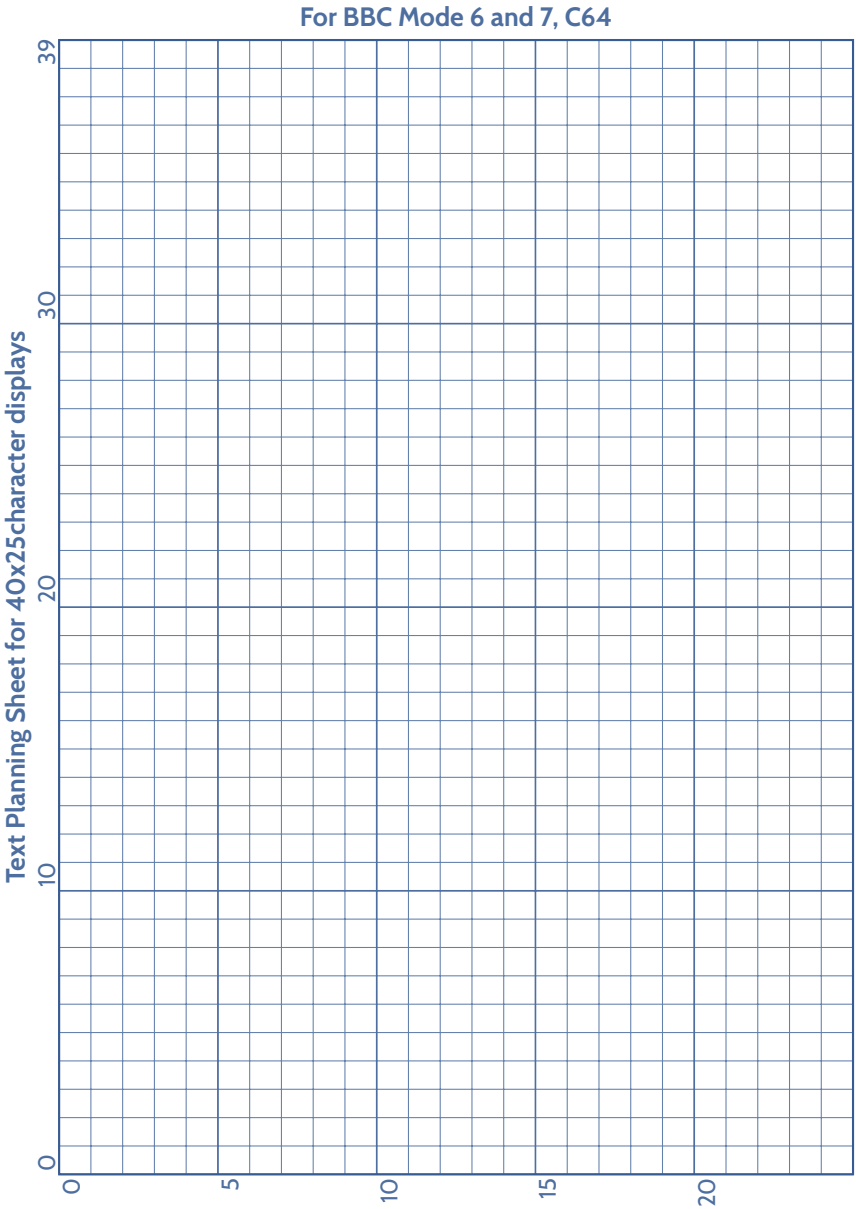

For BBC Mode 0
Text Planning Sheet for 80x32character displays
0
10
20
30
40
50
60
70
79
0
5
10
15
20
25
30

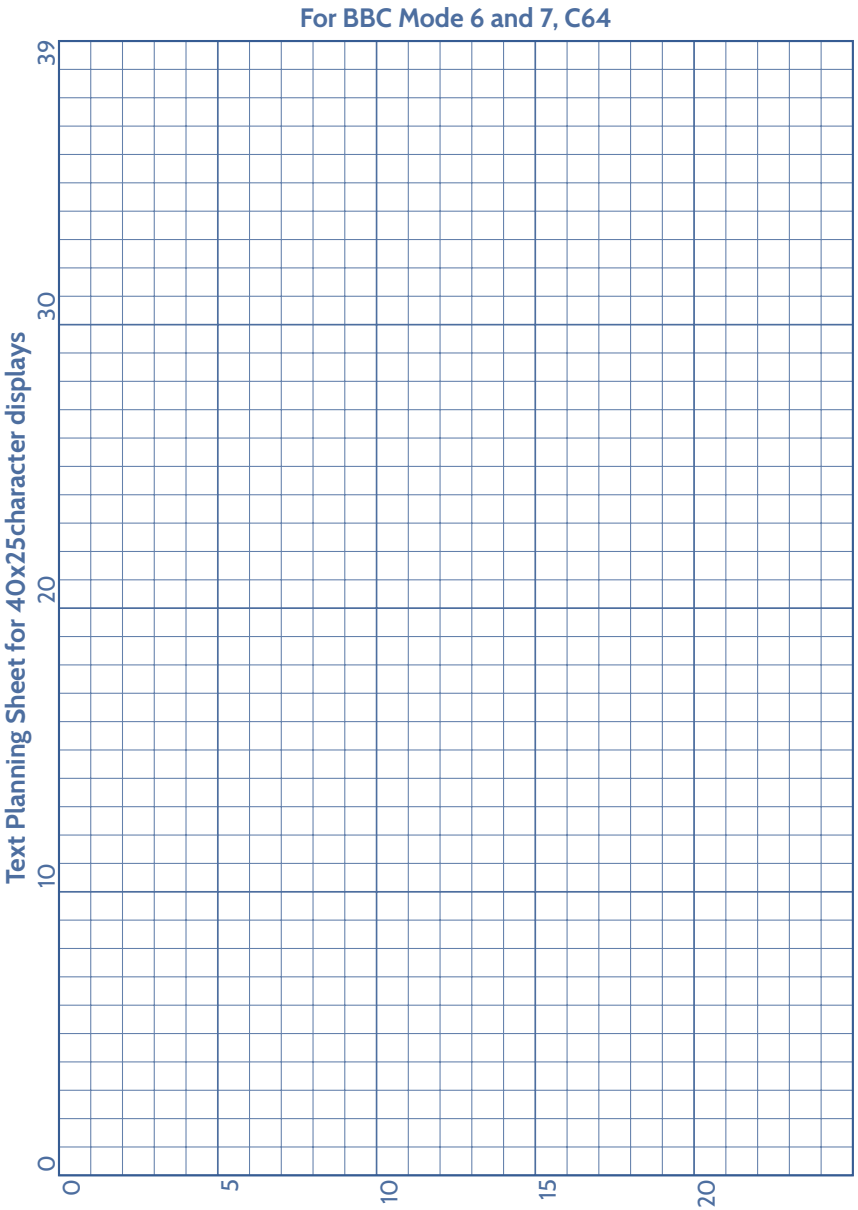
K.2 40x32 text screen planning sheet



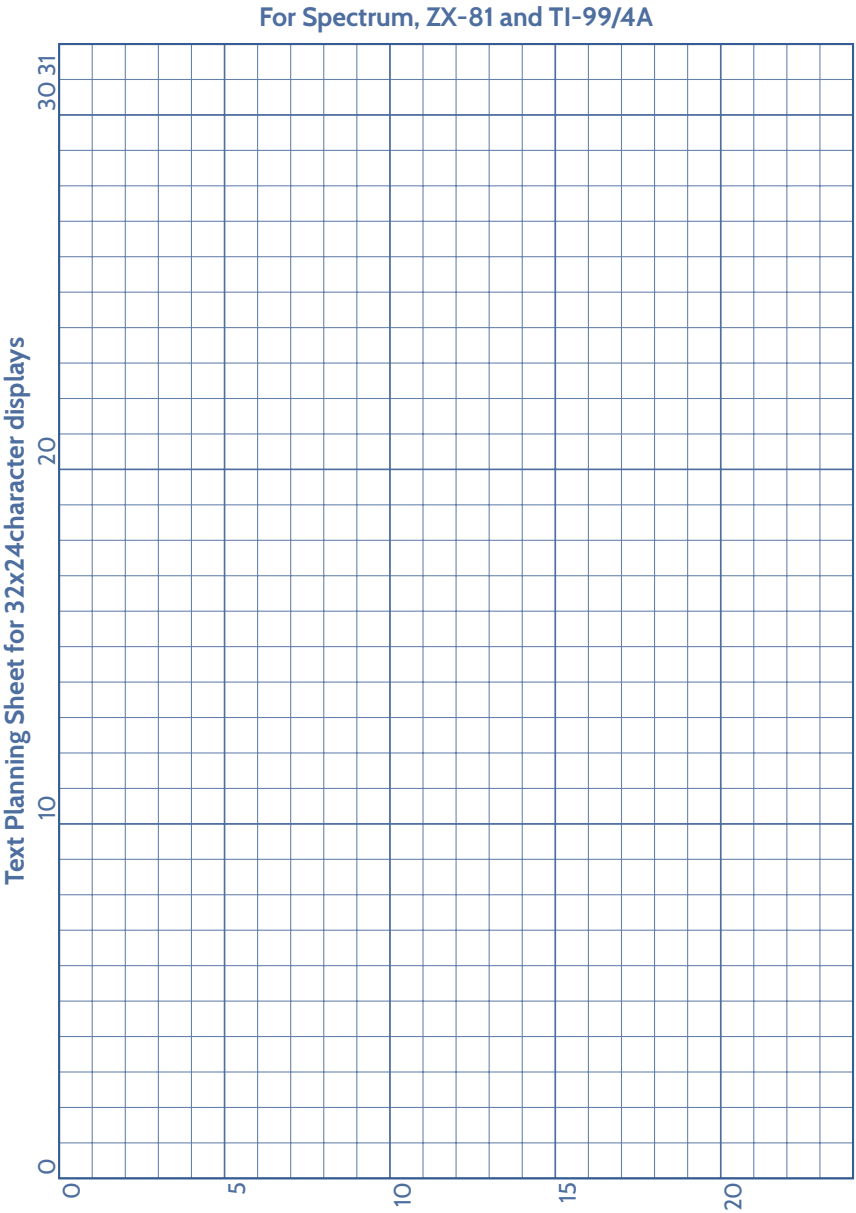

For BBC Mode 1 and 4
Text Planning Sheet for 40x32character displays
0
10
20
30
39
0
5
10
15
20
25
30

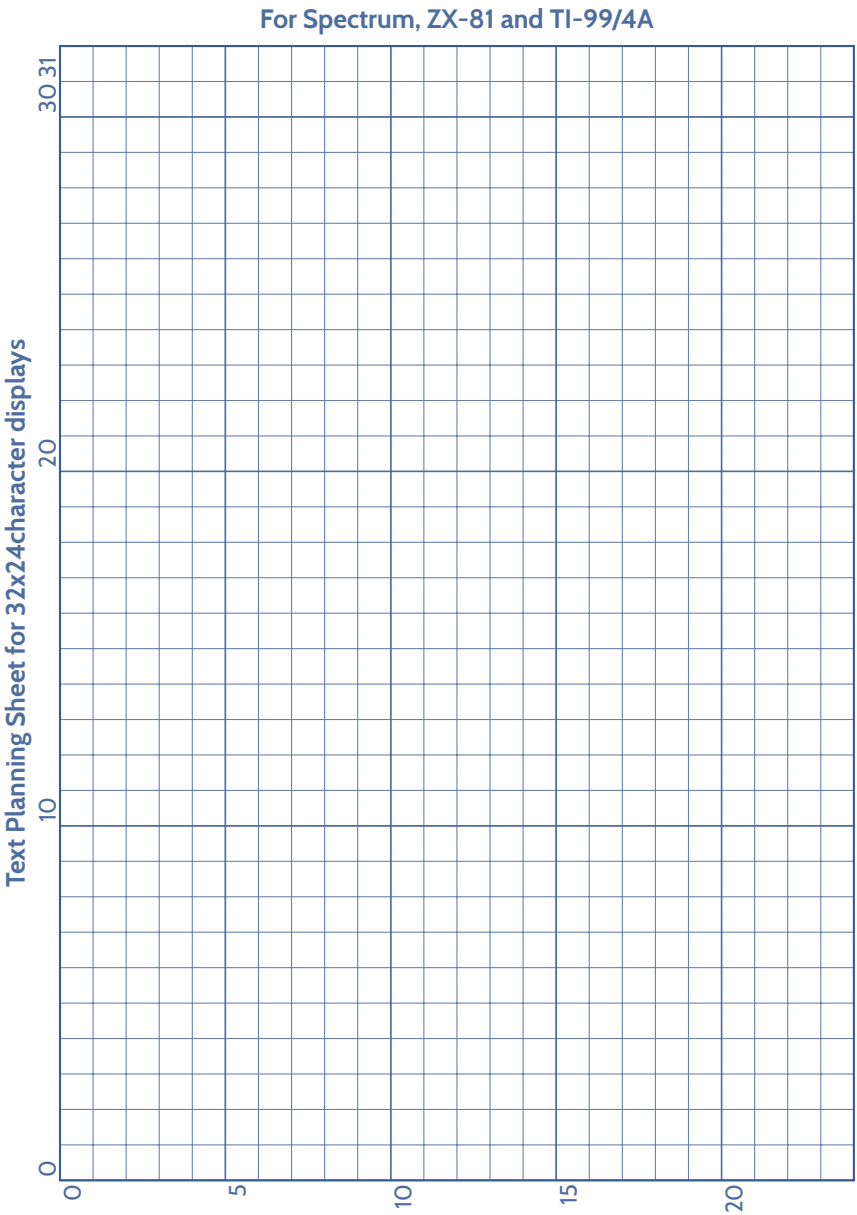
K.3 40x25 text screen planning sheet



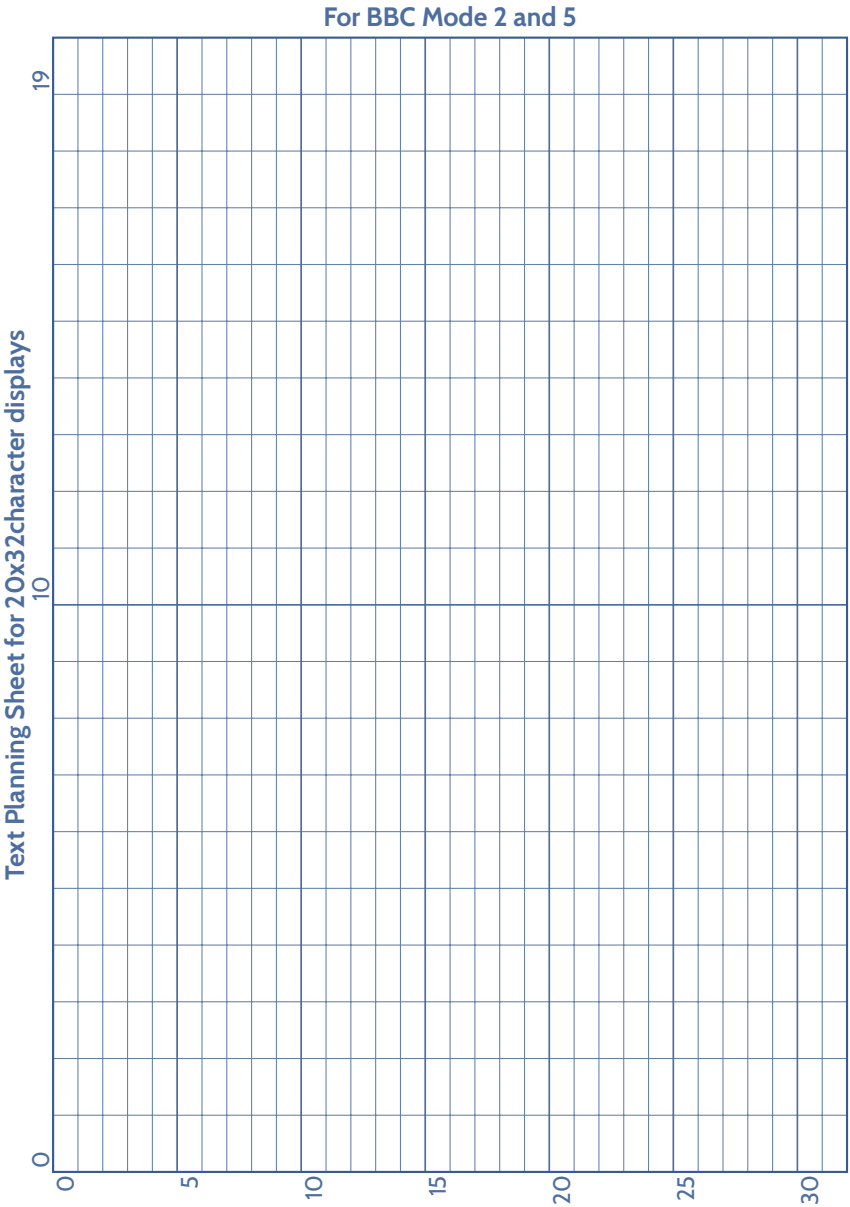

For BBC Mode 6 and 7, C64
Text Planning Sheet for 40x25character displays
0
10
20
30
39
0
5
10
15
20

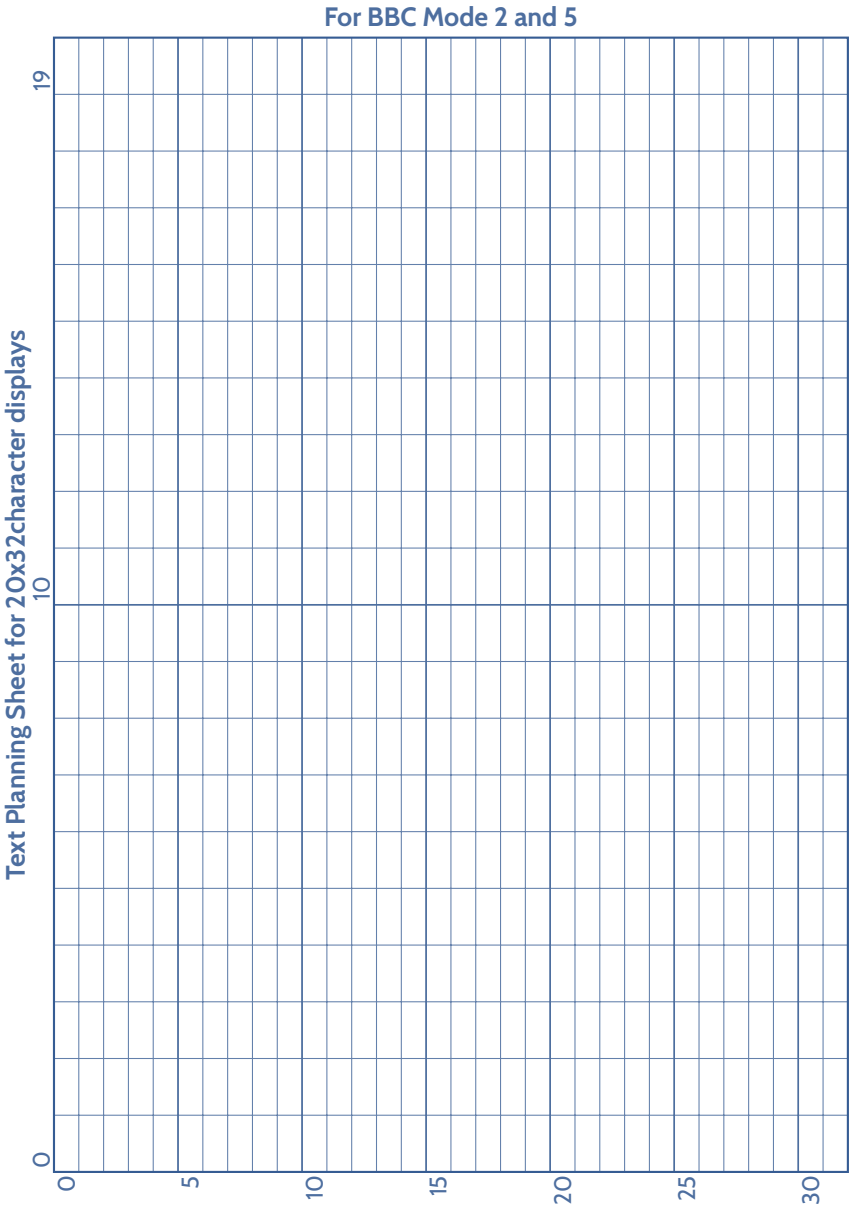
K.4 32x24 text screen planning sheet




For Spectrum, ZX-81 and TI-99/4A
Text Planning Sheet for 32x24character displays
0
10
20
30 31
0
5
10
15
20

K.5 20x32 text screen planning sheet



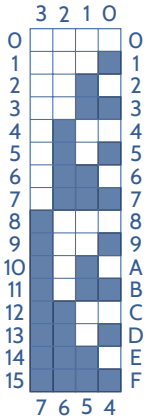

Text Planning Sheet for 20x32character displays
For BBC Mode 2 and 5
0
10
19
0
5
10
15
20
25
30

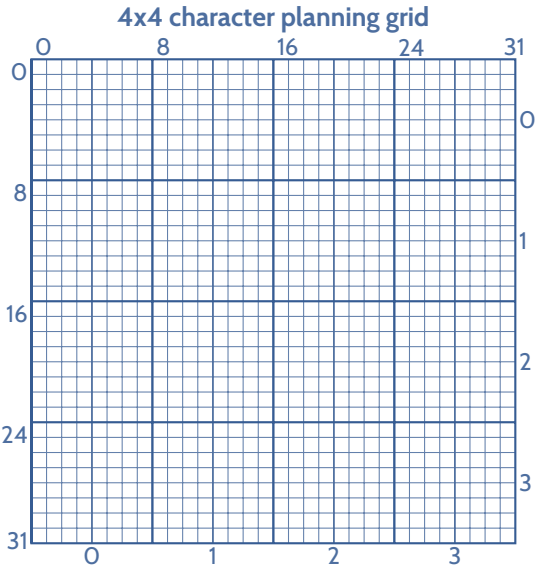
Page intentionally blank

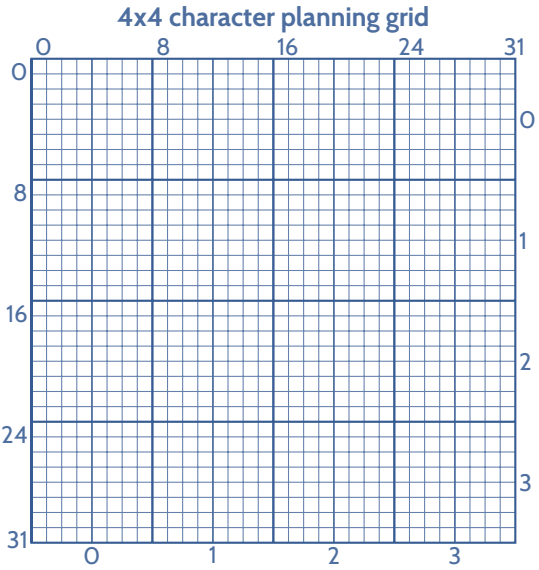
Graphics Planning Sheets

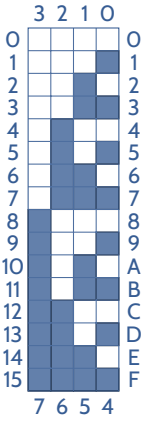
L

L.1 4x4 Character design sheet

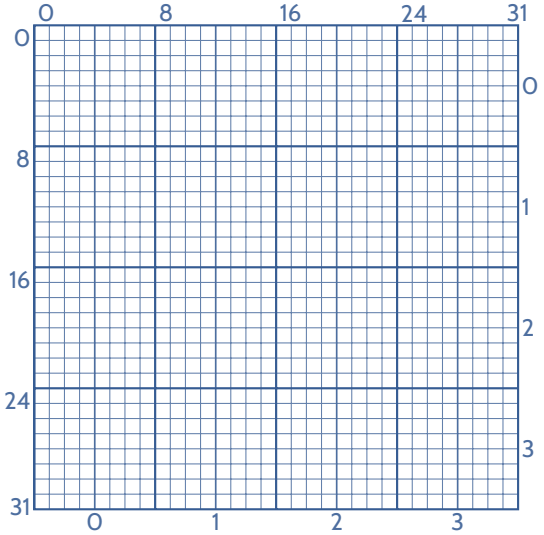




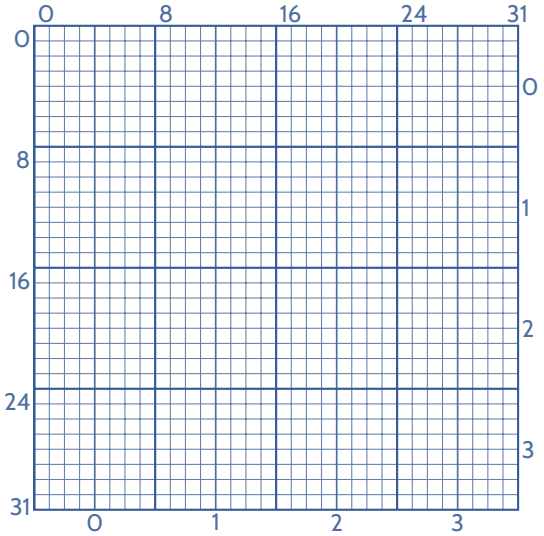




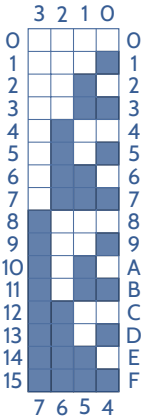
4x4 character planning grid

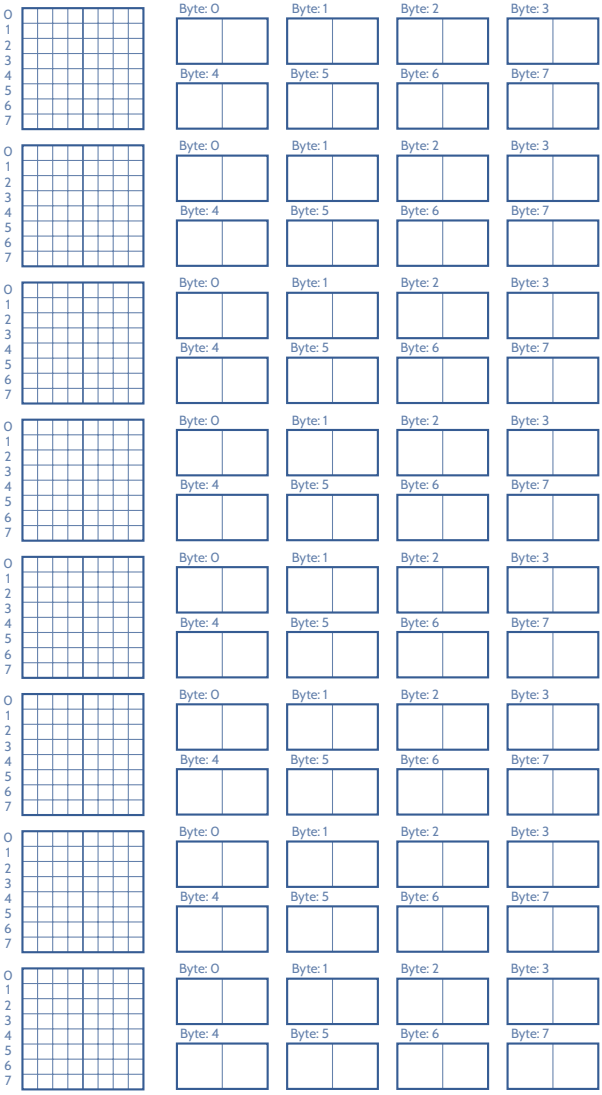


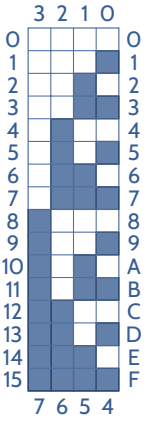
4x4 character planning grid

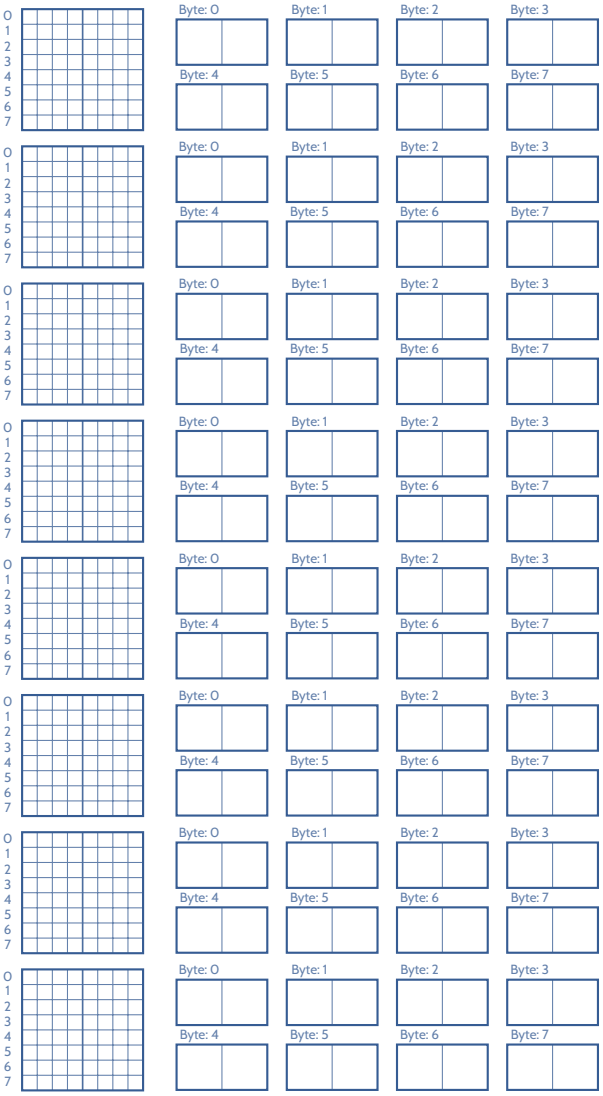


L.2 Character design sheet

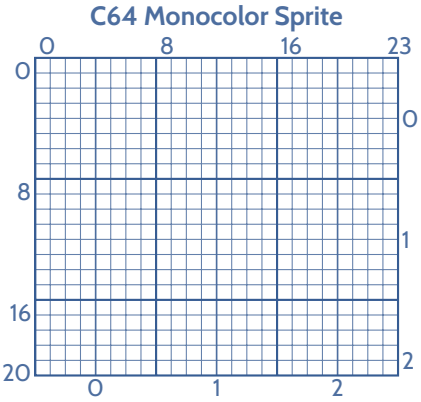


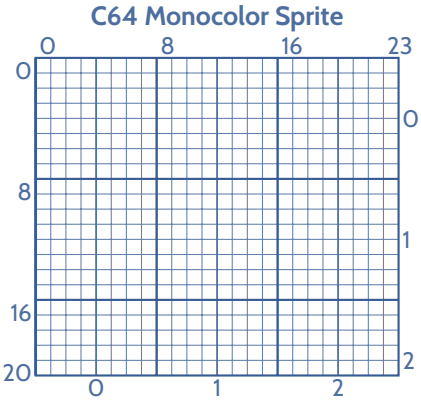


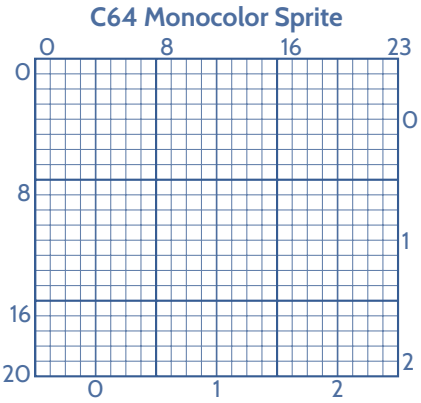

3 2 1 0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 1 2 3 4 5 6 7 8 9 A B C D E F
7 6 5 4

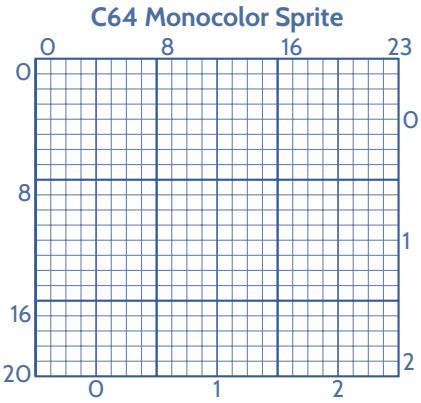

0 1 2 3 4 5 6 7
Byte: 0
Byte: 1
Byte: 2
Byte: 3
Byte: 4
Byte: 5
Byte: 6
Byte: 7

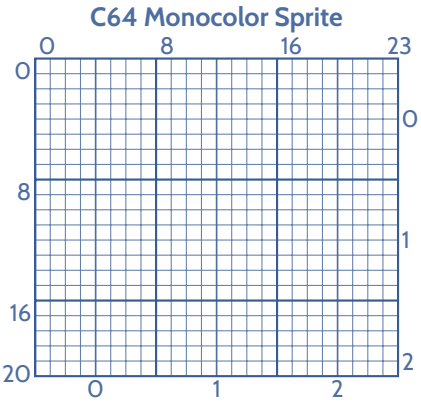
L.3 C64 Monocolour Sprite design sheet

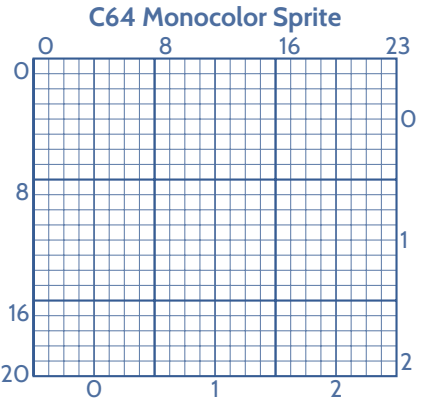




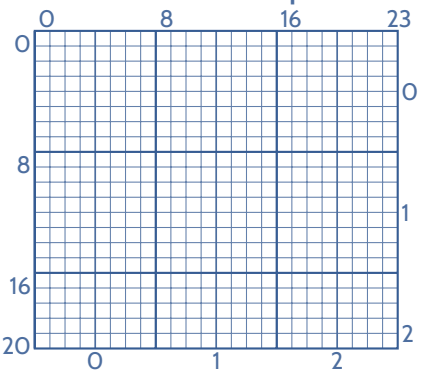




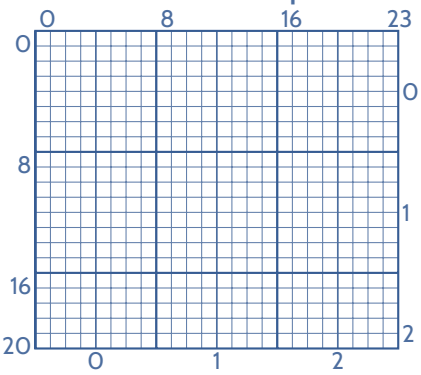




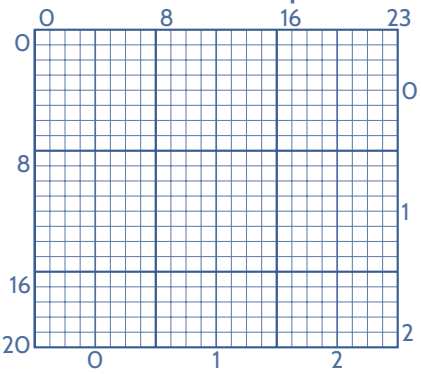
C64 Monocolor Sprite



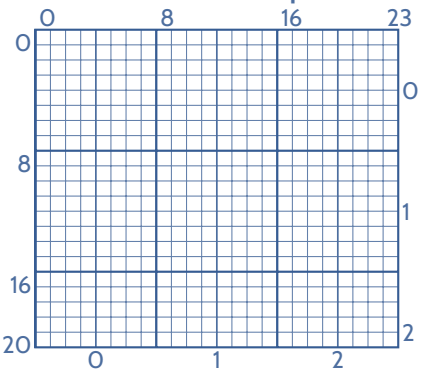
C64 Monocolor Sprite



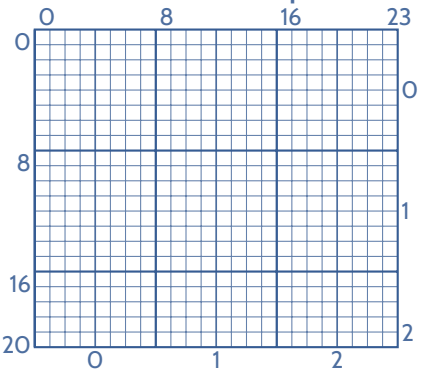
C64 Monocolor Sprite



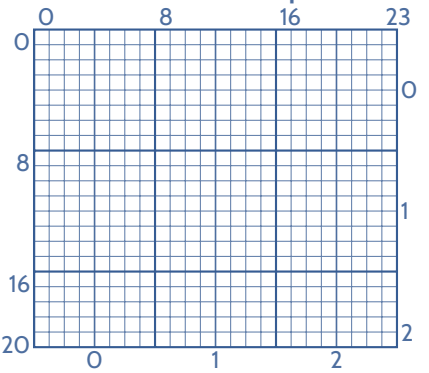
C64 Monocolor Sprite



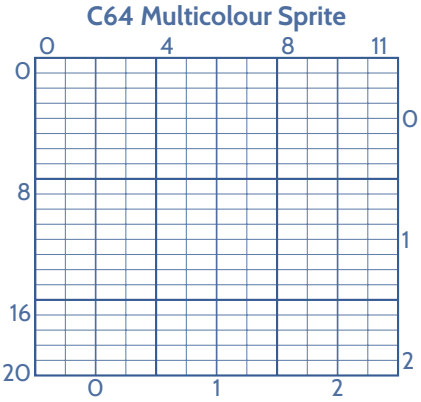
C64 Monocolor Sprite

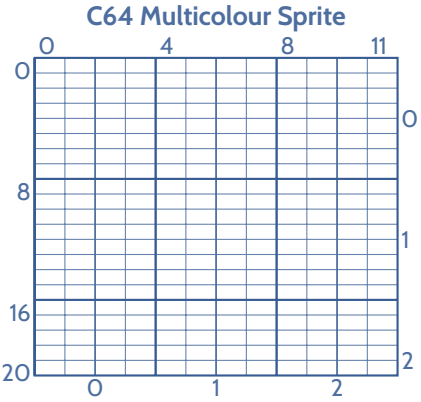


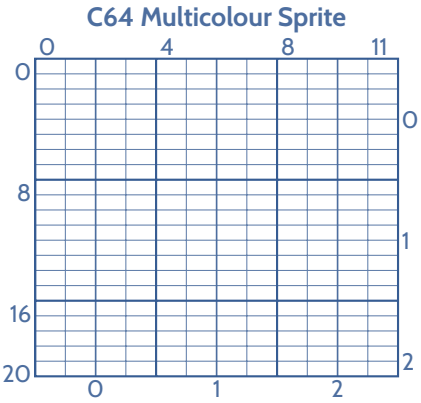
C64 Monocolor Sprite

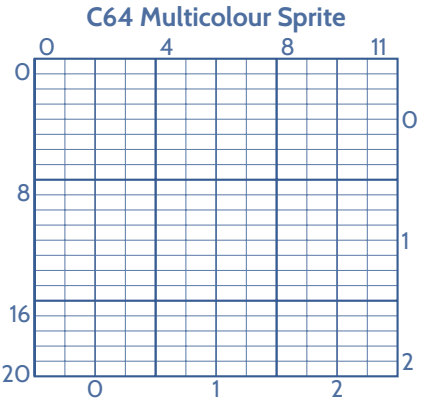


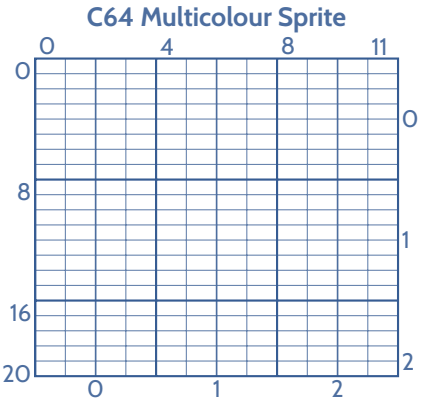
L.4 C64 Multicolour Sprite design sheet

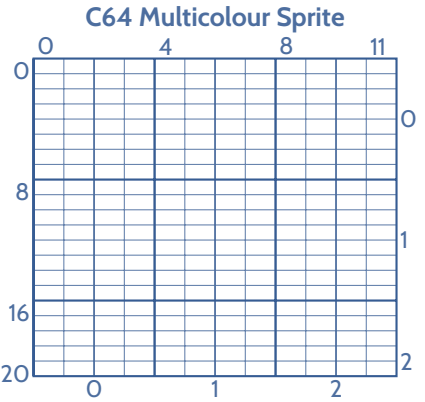




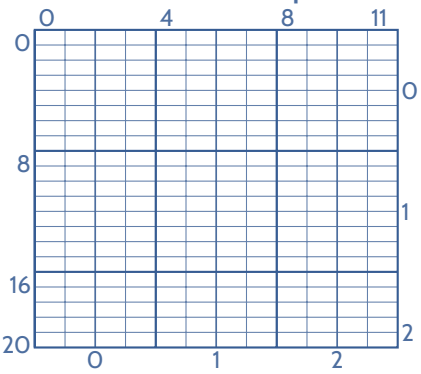




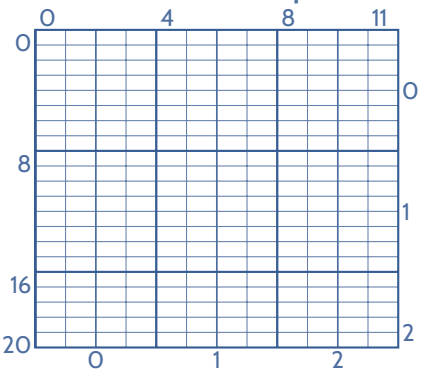




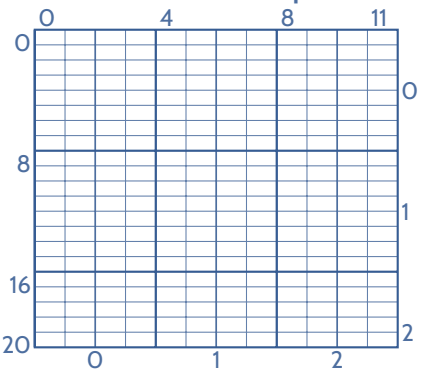
C64 Multicolour Sprite



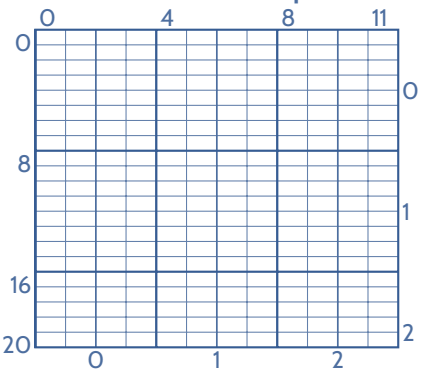
C64 Multicolour Sprite



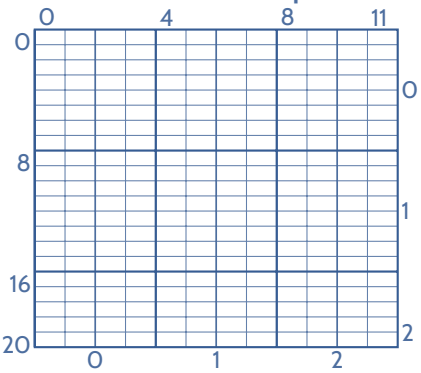
C64 Multicolour Sprite



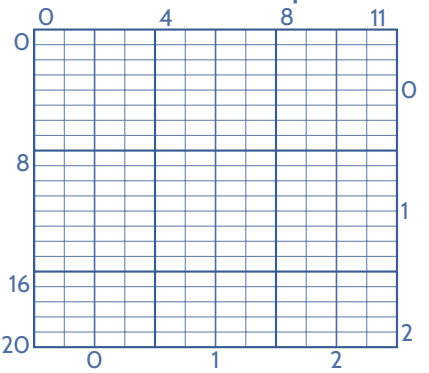
C64 Multicolour Sprite



C64 Multicolour Sprite



C64 Multicolour Sprite



Page intentionally blank

Indices

Page intentionally blank

List of Figures

7.1 Hexadecimal conversion chart 67
7.2 Binary representation of a byte 68
7.3 Capacity of multibyte values 69
7.4 Byte order 70
7.5 Two's complement representation of a byte 71
7.6 8-bit two's complement numbers 72
7.7 Capacity of multibyte two's complement numbers 72

10.1 AND truth table 91
10.2 Logical AND operation 91
10.3 OR truth table 92
10.4 Logical AND operation 92
10.5 Exclusive OR truth table 93
10.6 Logical Exclusive Or operation 93
10.7 Logical Complement operation 94

11.1 Rotate Left with Carry 98
11.2 Rotate Left with Branch Carry 98
11.3 Arithmetic Shift Left 98
11.4 Rotate Right with Carry 99
11.5 Rotate Right with Branch Carry 99
11.6 Arithmetic Shift Right 100
11.7 Shift left multiple 101
11.8 Rotate left multiple 101
11.9 Shift right multiple 102
11.10 Rotate right multiple 102

12.1 Decimal Multiplication 115
12.2 Binary Multiplication 115
12.3 Binary Division 117

16.1 AND truth table 136
16.2 Shift left 137
16.3 EOR truth table 165
16.4 Shift right 177
16.5 ORA truth table 181
16.6 Rotate Left 200

16.7 Rotate Right 201
16.8 **RTI** stack usage 203

17.1 Z80 CPU Registers 241

19.1 **NEG** and **CPL** 255

22.1 AND truth table 280
22.2 OR truth table 337
22.3 Stack before and after pop 346
22.4 Stack before and after push 348
22.5 Rotate Left 357
22.6 Rotate Left 359
22.7 Rotate Left with branch carry 360
22.8 Rotate Left with branch carry 362
22.9 Rotate Left Decimal 363
22.10 Rotate right 364
22.11 Rotate Left 366
22.12 Rotate Right with branch carry 367
22.13 Rotate right with branch carry 368
22.14 Rotate Right Decimal 370
22.15 Arithmetic Shift Left 382
22.16 Arithmetic Shift Right 384
22.17 Arithmetic Shift Right 386
22.18 Exclusive OR truth table 390

23.1 BBC Basic Assembler OPT values 406

26.1 BBC B Memory Map 565

27.1 BBC Master 128 Memory Map 569

29.1 Acorn Electron Keyboard 591
29.2 BBC B Keyboard 593
29.3 BBC Master 128 Keyboard 594

31.1 Anatomy of a $5\frac{1}{4}$" floppy disc 780

35.1 Acorn DFS Catalog Format 801

39.1 Duggan HDFS Catalog Format 829
39.2 Watford DFS Catalog Format 832

41.1 C64 Colour RAM layout 839

43.1 Spectrum 48K Keyboard 845

44.1 Spectrum Memory Map Overview 847
44.2 Memory/ROM paging port on 128K and +2 850
44.3 Memory/ROM paging port on Spectrum +2A and +3 in Normal mode 852
44.4 Memory/ROM paging port on +2A and +3 in Special mode 853
44.5 Spectrum Memory on +2A or +3 in Special Mode 854

45.1 TAP Format Header Block 856

F.1 BBC Micro RGB Port 999
F.2 BBC Micro RS423 Port 999
F.3 BBC Micro Cassette Port 1000
F.4 DIN5 cables compatible with BBC Cassette Port 1000
F.5 DIN3 cables compatible with BBC Cassette Port 1000
F.6 BBC Micro Analogue Port 1001
F.7 BBC Micro Econet Port 1002
F.8 BBC Micro Econet Port wired with the SPROW ethernet interface . 1002
F.9 Electron Master Cartridge Slot 1003
F.10 BBC Master Cartridge Slot 1004
F.11 Master Compact Cartridge Slot 1005

G.1 Commodore Serial Bus 1013
G.2 Commodore User port 1014
G.3 Commodore RS-232 port 1015
G.4 Commodore 64 Expansion Connector 1016

H.1 ZX-81 Edge Connector 1019

J.1 Spectrum 128K RGB Monitor Port 1023
J.2 Spectrum 128K RS232/MIDI Port 1024
J.3 Spectrum 128K RS232 AY-3-8912 Register 14 Bits 1024
J.4 Spectrum 128K Keypad Port 1025
J.5 Spectrum 128K Keypad AY-3-8912 Register 14 Bits 1025

J.6 Spectrum 128K Light Gun 1025
J.7 Spectrum 48K Edge Connector 1026
J.8 Spectrum 128K Edge Connector 1026

List of Tables

3.1 ADVAL(0) AND 3 21

6.1 Colours in TI Basic 53
6.2 Colour character sets in TI Basic 54
6.3 TI Noise Characteristics 57
6.4 TI Musical Frequencies 58

8.1 Status Flags by Processor 77

9.1 Conditional Branches by Processor 82
9.2 Conditional Subroutine Calls for the Z80 87

12.1 6502 Increment and Decrement instructions 103

13.1 6502 Vectors 122
13.2 65816 Vectors 122

16.1 Handling result from **CMP**, **CPX** or **CPY** 157

18.1 Z80 Flag Register 249
18.2 Half Carry Flag states 254

22.1 Z80 Instruction Format ADC 276
22.2 Z80 Instruction ADD 278
22.3 Z80 Instruction Format ADD 279
22.4 Z80 Instruction Format AND 280
22.5 Z80 Instruction Format BIT 281
22.6 Z80 Instruction CALL 284
22.7 Z80 Instruction Format CALL 284
22.8 Z80 Instruction CCF 285
22.9 Z80 Instruction Format CCF 285
22.10 Z80 Instruction CP 286
22.11 Z80 Instruction Format CP 287
22.12 Z80 Instruction CPD 289
22.13 Z80 Instruction Format CP__ 289
22.14 Z80 Instruction CPL 293

22.15 Z80 Instruction Format CPL 293
22.16 Z80 Instruction DAA 294
22.17 Z80 Instruction Format DAA 294
22.18 Adjustments to Accumulator performed by the DAA instruction . 295
22.19 Z80 Instruction Format DEC 297
22.20 Z80 Instruction DI 298
22.21 Z80 Instruction Format DI 298
22.22 Z80 Instruction DJNZ 299
22.23 Z80 Instruction Format DJNZ 299
22.24 Z80 Instruction EI 301
22.25 Z80 Instruction Format EI 301
22.26 Z80 Instruction EX 303
22.27 Z80 Instruction Format EX 303
22.28 Z80 Instruction EXX 304
22.29 Z80 Instruction Format EXX 304
22.30 Z80 Instruction HALT 305
22.31 Z80 Instruction Format HALT 305
22.32 Z80 Instruction IM 306
22.33 Z80 Instruction Format IM 306
22.34 Z80 Instruction IN 308
22.35 Z80 Instruction Format IN 309
22.36 Z80 Instruction Format INC 311
22.37 Z80 Instruction IND 313
22.38 Z80 Instruction Format IN__ 313
22.39 Z80 Instruction JP 317
22.40 Z80 Instruction Format JP 318
22.41 Z80 Instruction JR 319
22.42 Z80 Instruction Format JR 319
22.43 Z80 Instruction LD 322
22.44 Z80 Instruction Format LD 326
22.45 Z80 Instruction LDD 331
22.46 Z80 Instruction Format LD__ 331
22.47 Z80 Instruction NEG 335
22.48 Z80 Instruction Format NEG 335
22.49 Z80 Instruction Format NOP 336
22.50 Z80 Instruction OR 337
22.51 Z80 Instruction Format OR 338
22.52 Z80 Instruction OUT 342

22.53 Z80 Instruction Format OUT 342
22.54 Z80 Instruction OUTD 344
22.55 Z80 Instruction Format OUT__ 344
22.56 Z80 Instruction POP 347
22.57 Z80 Instruction Format POP 347
22.58 Z80 Instruction PUSH 349
22.59 Z80 Instruction Format PUSH 349
22.60 Z80 Instruction RES 350
22.61 Z80 Instruction Format RES 353
22.62 Z80 Instruction RET 354
22.63 Z80 Instruction Format RET 354
22.64 Z80 Instruction RETI 355
22.65 Z80 Instruction Format RETI 355
22.66 Z80 Instruction RETN 356
22.67 Z80 Instruction Format RETN 356
22.68 Z80 Instruction RL 358
22.69 Z80 Instruction Format RL 358
22.70 Z80 Instruction RLA 359
22.71 Z80 Instruction Format RLA 359
22.72 Z80 Instruction RLC 361
22.73 Z80 Instruction Format RLC 361
22.74 Z80 Instruction RLCA 362
22.75 Z80 Instruction Format RLCA 362
22.76 Z80 Instruction RLD 363
22.77 Z80 Instruction Format RLD 363
22.78 Z80 Instruction RR 365
22.79 Z80 Instruction Format RR 365
22.80 Z80 Instruction RRA 366
22.81 Z80 Instruction Format RRA 366
22.82 Z80 Instruction RRCA 367
22.83 Z80 Instruction Format RRCA 367
22.84 Z80 Instruction RRC 369
22.85 Z80 Instruction Format RRC 369
22.86 Z80 Instruction RRD 370
22.87 Z80 Instruction Format RRD 370
22.88 RST Restarts 372
22.89 Z80 Instruction RST 374
22.90 Z80 Instruction Format RST 374

22.91 Z80 Instruction Format SBC 376
22.92 Z80 Instruction SCF 377
22.93 Z80 Instruction Format SCF 377
22.94 Z80 Instruction SET 378
22.95 Z80 Instruction Format SET 381
22.96 Z80 Instruction SLA 383
22.97 Z80 Instruction Format SLA 383
22.98 Z80 Instruction SRA 385
22.99 Z80 Instruction Format SRA 385
22.100 Z80 Instruction SRL 387
22.101 Z80 Instruction Format SRL 387
22.102 Z80 Instruction SUB 389
22.103 Z80 Instruction Format SUB 389
22.104 Z80 Instruction XOR 391
22.105 Z80 Instruction Format XOR 391

23.1 BBC Basic Assembler OPT actions 407

24.1 Numeric operators 417
24.2 CPU options 422

25.1 Numeric operators 466
25.2 String operators 467
25.3 The 6502 compatible opcode mnemonics available 527
25.4 The 65C02-only mnemonics 527
25.5 Supported 6502 Address modes 528
25.6 Supported 65C02 Address modes 528

28.1 Default configuration for the BBC Master 572

29.1 Last Key Pressed 588

30.1 OS Provided OSCLI commands 601
30.2 Bit masks for OSBYTE calls $A \geq$ `0xA6` 604
30.3 OS versions 608
30.4 Input Streams 610
30.5 Output Streams 611
30.6 Cursor editing mode 612
30.7 Printer destination 613

30.8 RS-423 Baud rates . . . 614
30.9 Event types for OSEVEN . . . 617
30.10 Character definition expansion . . . 621
30.11 Buffer types for OSBYTE 0x15 . . . 622
30.12 Reset character definition ranges . . . 625
30.13 Sideways RAM allocation . . . 626
30.14 Sideways RAM Addressing . . . 626
30.15 Memory selection for OSBYTE 0x70 and 0x71 . . . 629
30.16 VDU Status . . . 632
30.17 Query Buffer Status . . . 640
30.18 *OPT for TAPE and ROM filesystems . . . 649
30.19 Cassette Speeds . . . 650
30.20 Current FRED allocations . . . 654
30.21 6850 ACIA control register bits 0...1 . . . 664
30.22 6850 ACIA control register bits 2...4 . . . 664
30.23 6850 ACIA control register bits 5...6 . . . 664
30.24 Speech processor commands . . . 667
30.25 Video ULA Palette Register . . . 685
30.26 ADC Conversion Type . . . 690
30.27 Master Compact ADC Conversion Types . . . 692
30.28 Master Compact Bit 5 return values . . . 692
30.29 Escape and Break Effect . . . 700
30.30 Keyboard Status . . . 701
30.31 Boot Options . . . 711
30.32 Boot Options . . . 716
30.33 Country Code . . . 734
30.34 Serial ULA Baud Rates . . . 737
30.35 Last Break Type . . . 744
30.36 OSByte 0xFE return . . . 745
30.37 Startup Options for OSBYTE `0xFF` . . . 750
30.38 Startup Drive timings - BBC Model B . . . 750
30.39 Startup Drive timings - Acorn Electron . . . 750
30.40 OSWord 00 parameter block . . . 753
30.41 MOS Clock . . . 755
30.42 Read/Write I/O Processor memory parameter block . . . 757
30.43 OSWord SOUND parameter block . . . 759
30.44 OSWord SOUND pitch on noise channel . . . 761
30.45 OSWord ENVELOPE parameter block . . . 763

30.46 OSWord Read pixel value parameter block 764
30.47 OSWord Read character definition 765
30.48 OSWord Read palette 766
30.49 OSWord Read palette 767
30.50 OSWord RTC Clock Parameters 771
30.51 OSWord RTC Clock BCD Parameters 771
30.52 OSWord RTC Clock set new time parameters 773
30.53 OSWord RTC Clock set new date parameters 773
30.54 OSWord RTC Clock set new date and time parameters 774
30.55 OSWord Z80 parameters 775

32.1 Acorn DFS *OPT4 parameters 785

35.1 Acorn DFS *OPT4 parameters 798
35.2 Boot action 803
35.3 Disk Type 803

44.1 Spectrum 128K/+2 Paging Port 849
44.2 Spectrum +2A/+3 Paging Ports 851
44.3 Spectrum +2A/+3 ROM ID's 851

47.1 List of fields in character-sets.xml 863
47.2 List of fields in processor-6502 dataset 865
47.3 List of fields in processor-z80 dataset 866
47.4 List of fields in screen-resolutions.xml 868

48.1 List of fields in acorn-mos.xml 871
48.2 List of fields in acorn-mos-osbyte.xml 874
48.3 List of fields in acorn-mos-osword.xml 876
48.4 List of fields in acorn-mos.xml 877

49.1 List of child elements for generate 890
49.2 Variables within hex-grid 897
49.3 Available Transforms 917
49.4 Available Transforms 917
49.5 List of child elements for generate 918

A.1 6502 Instructions by Opcode 921
A.2 6502 Instructions by Hex 928

B.1 Z80 Instruction Grids 941

C.1 Z80 Instruction prefixes 956
C.2 Generic bit types 957
C.3 Register types r, r' and s 957
C.4 Condition type c and C 958
C.5 Repeat and Direction bits C and D 958
C.6 Bit type b 958
C.7 Z80 Instruction Decode Table 959
C.8 Z80 Instruction Decode Table - CB PREFIX 960
C.9 Z80 Instruction Decode Table - DD PREFIX 961
C.10 Z80 Instruction Decode Table - ED PREFIX 962
C.11 Z80 Instruction Decode Table - FD PREFIX 963

D.1 MOS System Calls 965
D.2 OSByte calls 967
D.3 OSByte calls by action 970
D.4 OSWord calls 973
D.5 MOS zero-page 975
D.6 Tape filesystem options in 0x00BB 976
D.7 Filesystem Status byte 977
D.8 MOS Variables 978
D.9 MOS Variables by Name 980
D.10 Machine type returned by INKEY and OSBYTE 0x81 983
D.11 BBC Micro Keyboard codes 988
D.12 VDU 23,1 cursor states 992
D.13 BBC Master CMOS Ram 993

E.1 Common filesystem commands compatibility matrix 998
E.2 Filesystem selection command compatibility matrix 998
E.3 Definitons of compatibility columns 998

F.1 Acorn Electron Cartridge Slot 1003
F.2 BBC Master Cartridge Slot 1004
F.3 Master Compact Cartridge Slot 1005
F.4 $+5V$ Max Load 1006
F.5 Cartridge ϕ clock signal source 1007
F.6 ROM select latch address 1007
F.7 *CLOCK* frequency 1009

G.1 C64 User Port Pins 1014
G.2 C64 User Port Pins for RS-232 1015
G.3 C64 Expansion Port Pins 1016

I.1 Spectrum Keyboard codes 1021

K.1 Screen Text Resolutions 1027

1 RGBI CGA colour palette 1087
2 RGB colours 1107
3 Lifespan of the Sega Master System 1111

List of Listings

2.1 Sinclair Basic save program with auto execute 7
2.2 Simple guess the number 9
2.3 Simple guess the number with ELSE 9
2.4 Classic Hello World in Basic 10
2.5 Simple guess the number 10
2.6 Data example 1 11
2.7 Data example 2 11

5.1 Sinclair Basic For loop with nesting 27

6.1 LET in TI-Basic 41
6.2 LET in TI-Basic 42
6.3 ON GOTO example 43
6.4 CALL CHAR in TI-Basic 51
6.5 CALL GCHAR in TI-Basic 54

9.1 Conditional JSR on 6502 88
9.2 Conditional RTS on 6502 89

12.1 16-bit integer multiplication on the 6502 116

16.1 Implementing **PHX** on the 6502 190
16.2 Implementing **PHY** on the 6502 191
16.3 Implementing **PLX** on the 6502 197
16.4 Implementing **PLY** on the 6502 198
16.5 Switch 65816 to 6502 emulation mode 238
16.6 Switch 65816 to native mode 238

19.1 Example of equals and not-equals 257
19.2 Example of less than 258
19.3 Example of less than and greater than or equals 258
19.4 Example of less than or equals 259
19.5 Example of greater than 259

20.1 Scan keyboard 267

22.1 Example of DJNZ 300

23.1 Example 6502 assembly language program in BBC Basic 398
23.2 Example 6502 assembly language program in BBC Basic 402
23.3 Broken assembly language program with forward reference . . 403
23.4 Working assembly language program with forward reference . 404

24.1 Cloning BeebAsm from GitHub 410
24.2 Source Example 412
24.3 Local labels 413
24.4 Global label within local scope 414
24.5 Define label in parent scope 415
24.6 LO/HI example 418
24.7 Literals 420
24.8 FOR NEXT example 424
24.9 FOR NEXT with forward references 426
24.10 IF ENDIF examples 427
24.11 Macro example 429
24.12 Calling Macro's from Macro's 430
24.13 MAPCHAR example 432
24.14 Print Examples 433

25.1 Examples of valid labels 462

30.1 OSASCI source 597
30.2 Using OSBYTE 0x98 on the BBC B 659

44.1 Switching memory bank on a 128K Spectrum 849

48.1 Example of api synopsis 873
48.2 Example of OSBYTE synopsis 875
48.3 Example of OSWORD synopsis 876
48.4 Example of api entry 878

49.1 Example of char element 880
49.2 Example of font element 882
49.3 Example of defining an instruction format 886
49.4 Example of defining a formatType 889
49.5 Example of generate element 891

49.6 Example of hexgrid 898
49.7 Example of opcodeFigure 902
49.8 Example of opcodeList 902
49.9 Example of generating formats 905
49.10 Example of section element 909
49.11 Example of match within a field definition 917

Page intentionally blank

Bibliography

Here are the references in citation order.

[1] *Texas Instruments TI-99/4A Users Reference Guide*. Texas Instruments, 1982 (cit. on p. 35).

[2] *Z80 CPU User Manual*. UM008011-0816. Zilog Inc, Aug. 2016 (cit. on pp. 274, 372, 1121).

[3] Rodnay Zaks. *Programming the Z80*. 3rd Edition. Sybex, 1982 (cit. on p. 321).

[4] Holmes Dickens. *Adv. User Guide, BBC Master, Comp., B, B+, Electron*. Remastered by dv8 in 2017, Sixth revision June 2022. Adder Publishing Limited, June 2022 (cit. on pp. 526, 587, 592, 602, 607, 609, 610, 622, 640, 643, 648, 653, 667, 668, 671, 680, 686, 710, 728, 729, 746, 748, 749, 752, 758, 769, 770, 772, 992, 1006, 1007).

[5] Holmes Bray Dickens. *Adv. User Guide, BBC Micro*. Third edition Nov 1983. Cambridge Microcomputer Center, Nov. 1983 (cit. on pp. 526, 602, 622, 648, 653).

[6] *Master Series Ref. Manual Part 2*. Mar. 1986 (cit. on p. 526).

[7] *Support Group Application Note No. 203*. Support Group Application Note No. 203, Issue 1 dated 27th February 1992. Feb. 27, 1992 (cit. on p. 571).

[8] Holmes Dickens. *Adv. User Guide, Electron*. Second edition June 2008, first edition Sep 1984. Adder Publishing, Cambridge, June 2008 (cit. on pp. 587, 592, 607, 622, 648, 653, 668, 671, 678, 710, 728, 729, 746, 748).

[9] *Master Series Ref. Manual Part 1*. Jan. 1986 (cit. on pp. 607, 668, 749, 758, 769, 1009).

[10] *BBC Micro. User Guide*. Oct. 1984 (cit. on pp. 622, 640, 648, 653, 668, 761, 965, 992, 1027).

[11] beebwiki.mdfs.net. *OSBYTE 0x15*. Apr. 28, 2026. URL: http://beebwiki.mdfs.net/OSBYTE_%2615 (cit. on p. 622).

[12] beebwiki.mdfs.net. *INKEY*. May 10, 2026. URL: https://beebwiki.mdfs.net/INKEY (cit. on p. 643).

[13] beebwiki.mdfs.net. *OSBYTE 0x8A*. Apr. 28, 2026. URL: http://beebwiki.mdfs.net/OSBYTE_%268A (cit. on p. 648).

[14] beebwiki.mdfs.net. *OSBYTE 0x91*. Apr. 28, 2026. URL: http://beebwiki.mdfs.net/OSBYTE_%2691 (cit. on p. 653).

[15] *BBC Micro. Speech User Guide*. Issue 2, Part no 403950. May 4, 1983 (cit. on p. 666).

[16] beebwiki.mdfs.net. *Acorn DFS disk format*. Feb. 25, 2023. URL: http://beebwiki.mdfs.net/Acorn_DFS_disc_format (cit. on pp. 791, 827, 830).

[17] mdfs.net. *Acorn, Watford, Solidisk and Duggan DFS Filesystem Structure*. Feb. 25, 2023. URL: https://mdfs.net/Docs/Comp/Disk/Format/DFS (cit. on pp. 800, 828, 831).

[18] *Sprow Master 10/100 Net User Guide*. 2008 (cit. on pp. 813, 1002).

[19] *Commodore 64 Programmers Reference Manual* (cit. on pp. 841, 1013–1015, 1017).

[20] *Master Series Welcome Guide*. Mar. 1986 (cit. on p. 999).

[21] Steven Vickers. *ZX81 Basic Programming*. Second Edition 1981, reprint 1982. Sinclair Research Limited, 1981 (cit. on p. 1019).

[22] Wilson et al. *Spectrum 128 Rom Dissassembly*. Feb. 4, 2019 (cit. on pp. 1023–1026).

[23] Steven Vickers. *ZX Spectrum Basic Programming*. Second Edition 1983. Sinclair Research Limited, 1983 (cit. on p. 1026).

Glossary

default

+3DOS

The disk filing system for the Spectrum +3

Numbers

16-Bit

A computer who's internal architecture could manipulate data internally which was 16-bits wide. Usually this was also the width of the Data Bus but not always.

The 65816 is a 16-bit processor but only has a 8-bit Data Bus. The Z80 was a 8-bit processor even though it had some 16-bit instructions – the latter used a pair of registers, not a single one.

1977 Trinity

A set of three microcomputers from 1977 named by Byte as the "1977 Trility".

It was composed of the Apple II, CBM PET and the TRS-80.

3" floppy

A floppy disk with a semi-ridged case enclosing a 3" diameter disc. This version was not as popular as the 3.5" floppy.

32016

A 16-bit microprocessor by National Semiconductor.

32-Bit

A computer who's internal architecture could manipulate data internally which was 32-bits wide. Unlike earlier architectures like 16-bit, this was also the size of the Address Bus. These computers could handle a maximum of 4*Gb* of memory, although most OSs could only handle 2*Gb*.

3.5” floppy

A floppy disk with a semi-ridged case enclosing a 3.5” diameter disc.

3V3 **also written as** 3.3*V*

Most modern devices use 3.3*V* for both their power and logic levels. This makes them incompatible with retro microcomputers which used 5V.



Note:

Although this represents the voltage 3.3*V* it is usually printed as 3*V*3 as a safety measure because if a datasheet is copied multiple times the ‘.’ can be lost and suddenly 3.3*V* becomes 33*V* with the inevitable concequences.

5.25” floppy

A floppy disk with a flexible plastic case enclosing a 5.25” diameter disc.

5V

Most microcomputers used 5*V* or TTL for their logic levels and power supplies.

6309

The Hitachi 6809 is a CMOS version of the Motorola 6809 announced in 1985. In emulation mode it’s compatible with the 6809 but in native mode it’s 30% faster, has additional registers and 32-bit math.

64-Bit

A computer who’s internal architecture could manipulate data internally which was 64-bits wide, including the Address Bus. These computers could handle more than 4*Gb* of memory.

6501

An 8-bit microprocessor by MOS released briefly in 1975 which was pin compatible with the Motorola 6809 but with the core of the original 6502, meaning that compatibility with the 6809 was only at the hardware level.

The 6501 was discontinued after Motorola sued MOS.

6502

An 8-bit microprocessor originally released in 1975, which was popular with the early Microcomputers like the Apple 1, Apple II, BBC Microcomputer and C64.

The original 6502 part ran at $1MHz$.

Versions of this are still manufactured today and is used in microcontrollers.

6502A

A version of the 6502 used in Asteroids Deluxe which ran at $1.5MHz$, and on the BBC Microcomputer at $2MHz$.

6502AC

A version of the 6502 capanble of running at

$$3MHz$$

. This version was used in the Apple III and early Atari 8-bit machines, running at $1.8MHz$.

6502C

The official 6502C ran at up to $4MHz$. This is not the same chip as the 65C02 or the Atari SALLY.

6503

Version of the 6502 which could only address 4K of memory and no RDY, ϕ_1, SYNC, SO or redundant V_{ss} pins being omitted. Contained in a 28-pin dip.

6504

Version of the 6502 which could only address 8K of memory and no NMI, ϕ_1, SYNC, SO or redundant V_{ss} pins being omitted.

6505

Version of the 6504 which could only address 4K of memory. It lacked the same pins 6504 and was in a 28-pin dip.

6506

Version of the 6506 which could only address 4K of memory but had akll 3 clock pins were present. It was in a 28-pin dip package.

6507

Version of the 6502 which could only address 8K of memory. It had no support for interrupts and ϕ_1, SYNC, SO or redundant V_{ss} pins being omitted. It was used in the Atari 2600 as well as many Atari 8-bit peripherals.

6508

Version of the 6502 which contained 256 bytes of internal Static RAM and an 8-bit Input/Output port.

6509

Version of the 6502 which could address up to 1Mb of memory in 16 banks of 64K. It was used in the CBM-II.

6510

Version of the 6502 with a 6-bit Input/Output port. It was used in the C64. The 8500 replaced this chip in latter C64 machines.

6512

Version of the 6502 which relies on an external clock – the earlier processors had an internal clock generator. The 6512 was used in the BBC Model B+64 machine.

6513

Similar to the 6512 except this is extends the 6503.

6514

Similar to the 6512 except this is extends the 6504.

6515

Similar to the 6512 except this is extends the 6505.

65802

A discontinued 16-bit successor to the 65C02. Although it had a 65816 core, it only had a 16-bit Address Bus and fewer pins as it was intended as a pin-replacement to the 65C02.

65816

A 16-bit successor to the 65C02 which would run original 8-bit code but also had a 16-bit capability. It came with a 24-bit address bus enabling access to 16Mb of address space, but only a 8-bit Data Bus.

Machines like the Apple IIgs used this processor. This processor is still being manufactured.

65C02

A CMOS successor to the 6502 which fixed some bugs and introduced some new instructions.

65C102

A version of the 65C02.

68000

The Motorola 68000 is an 16-bit microprocessor. It was used in computers like the Macintosh 128K, Amiga and Atari ST.

6809

The Motorola 6809 is an 8-bit microprocessor with some 16-bit features, introduced in 1978. It was used in home computers, arcade games, and digital audio workstations, and was among the most powerful 8-bit processors of its era.

680X0

A family of 16-bit and 32-bit microprocessors by Motorola starting with the 68000.

The family was used across a wide area from the Sinclair QL, Macintosh 128K, Amiga, early Sun machines, calculators and even some space hardware.

Introduced in 1979, development was ceased by Motorola in 1994 when it was replaced with the PowerPC architecture. However, derivative processors remain and used in embedded systems.

6845 CRTC

A display controller by Motorola and originally intended for their 6809 cpu. However, it was used more on 6502 and Z80 based systems including the BBC Microcomputer.

68K

Abbreviation for the 680X0 family of microprocessors.

8" floppy

A floppy disk with a semi-ridged case enclosing an 8" diameter disc.

8080

A 8-bit microprocessor produced by Intel in the early 1970's. The Z80 was designed as a replacement for the 8080.

8080A

A version of the 8080.

8085

A version of the 8080.

8086

A 16-bit processor by Intel.

80186

A 16-bit processor by Intel, used in the BBC Master 512.

8500

A HMOS version of the 6510 with a 6-bit Input/Output port. It was used in the latter C64 machines.

8-Bit

A computer who's internal architecture could manipulate data internally which was 8-bits wide. These machines also had an 8-bit Data Buss.

A

Acorn **Acorn Computers Limited**

The manufacturer of the popular BBC Microcomputer family of computers including the Acorn Electron and the Acorn Atom.

They also designed the original ARM microprocessor now used on the majority of Tablet's and Mobile Phones in the world, along with small

boards like the Raspberry PI.

Acorn ADFS Advanced Disk Filing System

The successor to Acorn DFS which supported more files per disk as well as proper subdirectories. This mainly appeared on the Acorn Electron with the and the BBC Master 128 and was the primary filesystem for systems which had hard drives.

Acorn Atom

The first all-in one machine produced by Acorn which had the design features that would have been used in the Acorn Proton before that became the BBC Microcomputer.

Acorn Atom DOS

The Disk filing system for the Acorn Atom which was just the name for System DFS running on the Acorn Atom.

Acorn Business Computer

A business computer produced by Acorn based on the BBC Microcomputer.

Acorn Communicator

A business computer produced by Acorn with an lcd display and a built in modem. It used the 65816 microprocessor.

Acorn DFS Disk Filing System

The first filing system for floppy disks produced by Acorn, primarily used on 5.25" floppy disks. It supports a maximum of 31 files per disk and either 40 or 80 track drives.

Double sided disks were supported by having each side as a separate disk. *e.g.* Logical drives 0 and 2 are the two sides of the first physical drive and logical drives 1 and 3 for the second.

In modern times the format is used for disk images used with GoTek and other drive emulators.

Acorn Electron

A cutdown version of the BBC Microcomputer, originally intended to bring down the cost of children accessing the BBC Microcomputer family.

Acorn Proton

The planned successor to the Acorn Atom which would end up being released as the BBC Microcomputer.

Acorn System

An early computer produced by Acorn and had some of the features that would appear in their later 8-bit machines. It would be succeded by the Acorn Atom.

Active low

Signal which is active when it is at logic level 0, tied to ground or $0V$.

When written down, a line that is active low is shown with an $\overline{overline}$.

ADC Analogue to Digital Converter

A component which accepts an analogue voltage and provides a numeric value to the computer.

Address Bus

A set of connections which allows a CPU to address memory.

On 8-bit processors, the bus consists of 16 address lines allowing for 64K (65536) bytes of memory with addresses in the range of `0x0000` ...`0xFFFF`.

Some 16-bit processors used a 24-bit bus which allowed up to 16Mb of memory. 32-bit processors had a 32-bit bus which allowed up to 4Gb of memory, although most operating systems like Windows95 could only handle half of that.

AIM

A manufacturing aliance between Apple, IBM and Motorola. *See* PowerPC

All-in-one

A microcomputer which is in a single case containing both the motherboard and keyboard. Early machines like the C64, Spectrum and BBC Microcomputer are good examples of this style of case.

This is the opposite of the RM 380Z, TRS-80 or IBM PC where the keyboard was separate from the main computer.

Amiga Commodore Amiga

An extremely popular 16-bit and later 32-bit by Commodore based on the 68000 family of processors. The Amiga was one of the first home OSs which had true preemptive multitasking.

Amiga A1000

The first model of the Amiga range of microcomputers.

Amiga A1200

The sucessor to the Amiga A500 with an improved processor, more memory and updated chipset. It also had an IDE interface to install an internal 2.5" hard drive and a PCMIA slot.

Amiga A500

The first model of the Amiga that became popular to the home user. This machine is the one most people remember when they hear the name Amiga. This was an all-in-one machine unlike it's Amiga A1000 predecessor.

Accellerator cards or RAM expansions could be installed in a slot on the underside of the machine, whilst an expansion slot on the left hand side allowed for side-car expansions including hard drives, cd drives and RAM expansions.

Amiga A600

A cutdown version of the Amiga A500 which did not have the numeric keypad of it's Amiga A500 predecessor. It did have an IDE interface to install an internal 2.5" hard drive.

AmigaOS

A family of operating systems for the Amiga, specifically versions 1 through 3 for the Motorola 680X0 family of microprocessors. *See* AmigaOS 4 for V4.

The first version for the Amiga A1000 was released in July 1985.

AmigaOS 4

An operating system based on AmigaOS 3.1 and runs on PowerPC hardware.

Amstrad

Electronic's manuacturer who entered the computer scene with a series of microcomputers like the CPC464, CPC664 and CPC6128. They later bought Sinclair, influencing their later machines. They persisted into the early PC era.

Apple

A software and hardware giant, founded in late March 1976 with the launch of the Apple 1 computer. They later produced the popular Apple II system.

An early adopter of ARM in the early 1990's, they have been at the front of the computer industry for the last fifty years.

Apple 1

A single board machine based on the 6502 microprocessor. It was the first product produced by Apple.

Apple II

A system based on the 6502 microprocessor produced by Apple. It was their first machine which sold in bulk to the general public and was manufactured for about 10 years. It had the original version and the apple2e version.

A semi-portable version Apple IIc and Apple IIc+ was produced from 1984.

Apple IIc

A semi-portable version of the Apple II which had a built-in 5.25" floppy drive. It was not truly portable in that it had no internal screen and needed an external power source.

Apple IIc+

The successor to the Apple IIc which replaced the built-in 5.25" floppy drive with a 3.5" floppy drive. It was the last of the Apple 8-bit machines.

Apple IIe

A successor to the Apple II.

Apple IIgs

A 16-bit successor to the Apple II utilizing the 65816 processor.

Apple III

A successor to the Apple II. It had a similar layout to the Apple II except that it looked like the keyboard was separate – even though it was still permanently part of the main case.

Arcade

A dedicated machine which runs a game, contained in a custom enclosure which is usually the same height as the player so they can stand in front of it whilst playing. Some arcade machines allowed the player to sit on or in them. *e.g.* a driving simulator would have the player seated as if they were driving a real car.

These machines usually ran dedicated, custom hardware rather than a bespoke off the shelf computer. Modern machines are usually PC based.

see MiSTer

Archimedes

The first full computer powered by ARM produced by Acorn.

Arduino

A series of Open Hardware microcontroller boards.

ARM

A manufacturer and type of modern processors. These processors are in most mobile phones, tablets and small board computers like Raspberry PI.

The processors were orignally created by Acorn as the successor to the 6502 for their BBC Microcomputer successor, the Archimedes.

When ARM was spun out of Acorn, it was known as **Acorn Risc Machines**, as the processor was a type of RISC processor.

Later, ARM was renamed to **Advanced Risc Machines** to remove any link to Acorn who were in financial trouble at that point, before becoming the name now in use.

ARPANET

Advanced Research Projects Agency Network.

Arthur

An early operating system for ARM machines first released in 1987 with version 0.20, followed by 0.30 and 1.20. Arthur 2 became RISC-OS 2 in 1989.

ASCII American Standard Code for Information Interchange

A character set used by virtually every system since the microcomputer era.

Assembler

A program which takes Assembly Language for a specific CPU architecture and compiles an executable binary which that CPU can execute.

Assembly Language

A language which is used to represent the Machine Code of a computer. It is usually the lowest level language that can be implemented by a computer.

Atari

A former major games company who started with arcade games then moved to consoles and 8-bit machines before moving to machines like the Atari ST.

Atari SALLY

A custom 6502 microprocessor designed specifically for Atari and it's latter 8-bit machines as well as the Atari 5200 and 7800 consoles.

This has a $\overline{HALT}$ signal on pin 35 and the $R/\overline{W}$ signal on pin 36 – the normal 6502 chip has nothing on those pins. Pulling $\overline{HALT}$ low latches the clock pausing the processor, allowing the video circuitry access to the memory.

It is sometimes confusingly called the 6502C in Atari documentation – C being custom, although it was unrelated to the official chip with that number. Sometimes also labeled the C014806.

Atari ST

A range of 16-bit machines by Atari which had GEM as the graphical user interface. These machines were popular amongst musicians as they had MIDI ports as standard.

AY-3-8912

A programmable sound generator chip used in many devices including the Spectrum 128K.

B

BASIC **Beginners All-purpose Symbolic Instruction Code**

A simple programming language developed in the 1960's but was the default language on the majority of computer systems during the 1980's.

For the majority of programmers from that era, this was the first language they learned.

Battery Backed RAM

Static RAM of a type which used little power and used a battery so that it's content could be preserved when the main power was turned off.

Some systems like the BBC Master 128 used this to store it's settings, whilst others like the Acorn Electron had a cartridge which provided 32K as 2 16K banks of RAM which appeared as Paged ROMs.

Baud **Baud rate**

The rate at which a serial data link transfers data. One baud is equal to one bit of data transferred per second but is not always the same as bits per second.

BBC Basic

The flavour of BASIC which originally appeared on the BBC Microcomputer family of computers.

The ARM processors were actually first prototyoped in BBC Basic, primarily due to both it and the processor's Assembly Language were created by the same individual.

This form of BASIC was popular due to its use in UK schools and did

appear on other systems than Acorns.

Today it is still available for modern operating systems and rumour has it is still part of the test suite for ARM processors.

BBC Master 128

A successor to the BBC Microcomputer, which had additional memory and expansion options including Cartridge Slots and internal second processors.

BBC Master 512

A version of the BBC Master 128 with an internal 80186 second processor. The 512 in the name came from the second processor which had 512K of RAM - in addition to the 128K of the master.

BBC Master Compact

A compact, cutdown version of the BBC Master 128 which had the keyboard and base unit in separate cases similar to modern desktop machines. Hoever, unlike modern machines, the computer was in the keyboard with only the power supply and disk drives in the base unit.

BBC Master Econet Terminal

A version of the BBC Master 128 which was a terminal on an Econet network. Also known as the BBC Master 128 ET.

BBC Master Scientific

A BBC Master 128 with a 32016 coprocessor fitted to the internal Tube interface.

BBC Master Turbo

A BBC Master 128 with a 65C102 coprocessor fitted to the internal Tube interface.

BBC Microcomputer

A family of Microcomputers produced by Acorn, initially with the BBC Model B. Later the Acorn Electron and the BBC Master 128 followed.

BBC Model A

A cutdown version of the original BBC Microcomputer, which had 16K of memory and fewer peripheral ports. This machine could be upgraded into a full model B with the missing components being added to the

motherboard.

BBC Model B

The original BBC Microcomputer, which had 32K of memory. The BBC Model A was released at the same time which had only 16K of RAM and many of the ports were not populated.

BBC Model B+

Successor to the BBC Model B and an intermedary model before the BBC Master 128.

BCD Binary Coded Decimal

A form of arithmetic which uses each Nibble in a byte to store a single decimal digit.

BeOS

A discontinued multitasking graphical operating system.

Big-Endian

An alternate form of Endianess where the the most significant byte is first, with each lesser significant byte following it until the least significant byte is reached. *see* Little-Endian

Binary

A number system which is base 2 in mathematics. It consists of just the digits 0 or 1.

Bit

The fundamental unit of a computer's memory. It can hold either 0 or 1. *see* binary

Bluetooth™

A short-range wireless technoloty standard

Brandy Basic

An Open Source version of BBC Basic. Matrix Brandy Basic is a fork aiming to improve BASIC VI compatibility.

BSD

BSD *License*

A type of Open Source license.

BSD *Operating System*

A former OS based on Unix which evolved into FreeBSD, NetBSD and OpenBSD.

BT **British Telecom**

The primary telecomunications company in the UK during the late 1980's post privitisation. It was part of the Post Office prior to that. It covered most of the UK except for the city of Kingston-upon-Hull.

BT Merlin Tonto

An office machine based on the Sinclair QL, containing a proper mecanical keyboard, two Microdrive drives and a telephone handset on the base machine. It was sold by BT as part of their Merlin telephone system for operator and office use.

Byte

A microcomputer magazine published from September 1975 to July 1998 and was influential in the early microcomputer era.

Byte

8 bits of memory. Data is, on an 8-bit Microcomputer, transferred over the Data Bus one byte at a time.

C

C128 **Commodore 128**

The direct successor to the Commodore C64 based on the 6502. As the name indicates, this machine had 128K of RAM but it also had a Z80 processor along side it's 6502 enabling it to run alternate operating systems like CP/M.

C64 **Commodore 64**

An extremely popular system by Commodore based on the 6502 which was released in 1981. It was, until the Raspberry PI in the 2010's, the most popular single model machine ever produced.

Cambridge Co-Processor

An updated and successor to the 32016 second processor.

Cambridge Workstation

A range of workstations which succeded the Acorn Business Computer.

Carry Flag

In Assembly Language the Carry flag is used in arithmetic to carry over an overflow from an addition, or to borrow from a subtraction. In Shifts and Rotates it is used to pass a bit moved out of a register into another one.

Cartridge

A device which plugs into a computer or console to provide a game or application. On some systems this was the only form of running software. On others like the C64 a cartridge usually replaced the OS or BASIC.

Others cartridges could also provide hardware additions. *e.g.* the Acorn Electron cartridges could provide additional hardware ports.

Cassette

A type of tape media enclosed in a plastic case. It was originally used for holding audio so was one of the first forms of media used by the hobbyist of the 1970's and hence the early computers of the 1980's.

Cassette deck

A device which used cassettes to play music. Early microcomputers used them to store programs before the floppy drive became more common.

CBM PET

The PET was one of Commodores early systems.

CBM-II

A short lived successor to the CBM PET, introduced in 1982 but discontinued in 1984.

CD Drive

A device for storing data on an optical disk. It can hold 70 minutes of audio or 720Mb of data. These drives did not appear until the late 1980's and no 8-bit machine had them.

Centisecond

$10^{-2}s$ or 100th of a second. SI Unit **cs**.

The of many microcomputers like the BBC Microcomputer ran their clocks so that time incremented in $1cs$ steps. The main reason for this was due to the clock being synchronized to the frequency of displays which was in turn synchronized to the local mains electricity frequency.

e.g. in the UK the mains frequency ran at $50Hz$ which meant that each cycle ran for $2cs$.

Centronics

A family of connectors, mainly known for connecting to parallel devices like printers and SCSI devices like scanners.

CGA Color Graphics Adapter

An analog video graphics standard introduced with the IBM PC in 1981 providing 2 text and 3 colour display modes, utilizing a colour palette of 16 colours. Text was in either 40×25 or 80×25 character modes – the former intended for home use with a TV. Graphics were either 160×100 with 16 colours, 320×200 with 4 colours or 640×200 2 colours. It was superseded by EGA.

CISC Complex Instruction Set Computer

A type of instruction set for CPUs which have an extensive number of instructions, or have many instructions which could be implemented with many simpler ones. Processors like the Z80 and 8080 are known as CISC.

Although the 6502 is also CISC, it is not that far away from RISC due to the simplicity of it's instruction set.

Clock

A timing signal inside a computer to which all operations are based on. These clocks usually run at $1Mhz$, $2Mhz$ or more, depending on the system.

Table 1: RGBI CGA colour palette

Dec	IRGB	Colour
0	0000	Black
1	0001	Blue
2	0010	Green
3	0011	Cyan
4	0100	Red
5	0101	Magenta
6	0110	Brown
7	0111	Gray1
8	1000	Gray2
9	1001	Light Blue
10	1010	Light Green
11	1011	Light Cyan
12	1100	Light Red
13	1101	Light Magenta
14	1110	Light Yellow
15	1111	White

CMOS Complementary Metal Oxide Semiconductor

A chip manufacturing process which produces chips that use less power than the earlier NMOS process.

Commodore

A former American manufacturer who produced popular machines like the CBM PET, Vic20, C64 and the Amiga.

Their 8-bit machines were based on the 6502 primarily due to cost as they purchased MOS in the 1970's for that purpose.

Commodore Basic

The flavour of BASIC for the C64 by Commodore, although it first appeared on the CBM PET and Vic20.

Composite Video

An analogue video standard which contained both the V-Sync and video picture in a single signal. It supported both Monochrome and Colour displays by optionally including the colour component within the same

signal.

Many systems like the BBC Microcomputer had monochrome Composite Video outputs but could, either by a link or additional components, output a colour signal. The reason they did this is due to the colour component causing interference with the monochrome component so when used with monochrome monitors the signal was poorer than a pure monochrome signal.

Many later machines simply provided a colour Composite Video output.

The signal is similar to RF in that the modulators used to generate a RF signal usually used a Composite Video signal as their input. On some machines it was possible to tap into this to provide a Composite Video compativle signal – although proper circuits are available to do this replacing the modulators.

CP/M

A popular OS for Z80 based systems. PC-Dos and MS-Dos are related successors.

CP/NOS

A networked version of CP/M where the operating system was on a central fileserver rather than the local system. This was used on the RM 480Z.

CPC464

A popular 64K machine from Amstrad based on the Z80. It had a cassette deck built into the machine as well as a proper keyboard.

The machine came with either a Black and White or Colour monitor which was also the systems power supply.

CPC6128

A version of the CPC464 but with a built in floppy drive and 128K of memory

The machine came with either a Black and White or Colour monitor which was also the systems power supply.

CPC664

A version of the CPC464 but with a built in floppy drive.

CPU **Central Processing Unit**

The core component of a computer system which processes information. In Microcomputers this is a single chip at the heart of the computer.

In earlier generations the CPU was not a discrete component. For Minicomputers it was generally an entire board of components. Going even earlier then the CPU would be a series of boards to entire cabinets in size.

CRT **Cathode Ray Tube**

An analogue display formed of an electron gun aimed at a phosphor layer which displays a colour when the beam hits it, forming a display. No longer produced as it's been replaced cheaper and more energy efficient displays, it was the predominant display type for both computers and TV for over 50 years.

CTS **Clear To Send**

A line on a RS-232 or RS-423 serial cable which the receiver indicates the sender that it's clear to send data. The opposite end for this line is RTS.

D

Data Bus

A set of connections over which all data is transferred between devices. The number of lines on the bus is usually the size of the CPU. *e.g.* for 8-bit microcomputers the data bus is 8-bits wide, and in general 16-bit computers use 16-bits for the address bus.

Decimal

The Base 10 number system which uses the digits 0 … 9.

Digital **Digital Equipment Corportaion**

A former computer manufacturer known for their minicomputer systems like the PDP-11.

Digital Research

A software manufacturer who wrote PC-Dos and other software.

Dip **Dual Inline Pin description**

Double

A value which is double the size of the normal type. For bytes, a double commonly defines two bytes or a Word of memory. In floating point a double is a floating point value which has twice the precision of a normal float.

DRAM **Dynamic-RAM**

A form of RAM which has a higher density than Static RAM. Each bit is formed of a transistor and a capacitor. As this capacitor discharges over a short period of time, the memory needs to be refreshed frequently.

Driver

Software that enables the use of a device within a system.

DSub

A type of connector used to connect devices to a computer. The D in the name is because the connector looks like an uppercase D character.

Duggan HDFS

A disk filing system partially compatible with Acorn DFS.

E

Econet

A wired network of BBC Microcomputers. The cabling and protocol was proprietary to Acorn.

EEPROM Electrically Erasable Programable Read Only Memory

A type of ROM which can be programmed multiple times. They can be erased by the programmer by applying an appropriate voltage to the chip. Modern ROMs are usually of this type.

EGA Enhanced Graphics Adapter

The successor to CGA, this video graphics standard was introduced in 1984. It provided multiple new modes in addition to those in CGA. It was superseded by MCGA and VGA standards in 1987, although the latter would support the EGA modes.

Endianess

The order of bytes used to represent a multibyte value in memory within a computers' architecture. There are two types, the common Little-Endian and the less common Big-Endian.

EPROM Erasable Programable Read Only Memory

A type of ROM which can be programmed. Before an EPROM can be reprogrammed it has to be erased with U.V. light as they had a clear window on the top of the chip. You can usually identify EPROM's as they had a paper sticker covering the window.

ESP32

A family of 32-bit microcontrollers with Wi-Fi™ and Bluetooth™ capabilities.

Eurocard

An IEEE standard format for printed circuit board (PCB) cards that can be plugged together into a standard chassis which, in turn, can be mounted in a 19-inch rack.

F

File handle

A number which represents an open file. On many microcomputers this is a single byte value whilst on others an integer.

File System

The format of some media which contains files and, in more recent File Systems, directories.

Firm key

The combination of pressing the `CAPSLK/FUNC` key with the other keys on the Acorn Electron keyboard to emulate the normal function keys which the Electron does not have.

Firmware

Software resident in a device which controls it. Many devices are computers in their own right so their firmware is effectively their OS.

Floppy controller

A chip which manages a floppy drive.

Floppy disk

A disk containing a magnetic disk enclosed inside a flexible plastic case. They are usually in the sizes of 3.5", $5\frac{1}{4}$" in diameter although rarer sizes include 8" and 3".

Floppy drive

A device which uses either 5.25” floppy or 3.5” floppy disks. Some machines manufactuyed by Amstrad used the rarer 3” floppy disks, whilst the first drives used 8” floppy disks.

FreeBSD

An OS based on BSD and Unix. MacOS is based on the user land but uses a different kernel. *See* NetBSD, OpenBSD

G

GEM

GEM was an early form WIMP interface that first appeared on the Macintosh 128 and then the Atari ST also had GEM. Some other systems also had GEM style interfaces including the BBC Master 128 when running PC-Dos.

Genlock

A circuit which would synchronize two video signals enabling one to be displayed over the other. This allowed for some machines like the Amiga to be used to create graphics for live television.

GNU **GNU is Not Unix**

GoTek

A hardware emulator of a floppy drive which, with the appropriate firmware can replace a floppy drive with a USB device for storage.

GPIO **General Purpose Input Output**

Connections to a processor or device to which the user can interface external hardware to.

H

Handshake

Part of a communication protocol, where two sides notify each other they are ready to transfer data. In the case of RS-232 serial communication, this is done using the various handshaking lines like CTS and RTS.

Hard drive

A device for storing data for fast access. Like a floppy drive, a hard drive stores data on a rotating disk. However, it has far larger capacities than a floppy.

At first, they were far too expensive for hobbyist or home use until the end of the 1980's when prices came down to reasonable levels.

HDMI

A display interface protocol and cable for modern digital displays including TV's.

Hexadecimal

The Base 16 number system which uses the decimal digits $0 \ldots 9$ followed by $A \ldots F$.

e.g. `0123456789ABCDEF`

The value `0xF` is followed by `0x10`.

On some really old systems, *e.g.* 1950's and early 1960's, the letters are at the far end of the alphabet.

e.g. `0123456789UVWXYZ`

It fits in exactly 4 bits or one Nibble of memory.

High

On an electrical bus a high signal usually designates a 1. That is, on a 5*VTTL* system, if a line is at 5*V* then it is usually indicating a 1. The Address Bus and Data Bus of a microcomputer uses this level of signalling.

Note that for active low buses, a high value represents a 0.

Hitachi

A Japanese manuacturer

HMOS High density, short channel MOS

A form of depletion-load NMOS but used a single power supply voltage. This process was replaced by CMOS in the 1980's, but is still used today.

I

IBM International Business Machines

A major international company which produces mainframe computers. In the early 1980's they developed the IBM PC which became the blueprint for most modern computers.

IBM PC

A PC produced by IBM as a means for them to enter the home and small business computer market. The design was then cloned by other manufacturers and the overal design became the standard, ending the microcomputer era.

IBM PC-Jr

A basic IBM PC produced by IBM which had BASIC as it's default language and had the ability to use a cassette deck.

IDC Insulation Displacement Connector

A style of connector which uses a ribbon cable and when crimped to the connector will cut into the insulation to make an electrical contact.

IDE

A parallel interface for connecting hard drives and cd drives to computers using a ribbon cable. It was common in early PCs and could host one or two devices per cable. Some non-PC's like the Amiga used this interface for hard drives.

IEEE **Institute of Electrical and Electronics Engineers**

An American professional organization founded in 1963 for Electrical Engineering, Electronics Engineering and releated disiplines.

Intel

A major semiconductor and processor manufacturer who produced the first microprocessors.

J

Jiffy

Dictionary definition: A very short space of time.

In computing, a jiffy is the time one cycle of the local mains electrical supply. In the UK that is $\frac{1}{50}^{th}$ of a second as the electrical supply has a frequency of $50Hz$. In the US it is $\frac{1}{60}^{th}$ of a second for the same reason.

K

Kempston

A manufacturer of peripherals for many systems, commonly known for the joystick interface for the ZX-Spectrum.

Kernal

The name of the OS for the Commodore 8-bit systems, *e.g.* the CBM PET, Vic20 and C64.

Note the spelling is with an A not an E - this was probably a typing error made in the 1970's and the name stuck.

Kernel

In an OS, a kernel is the core, consisting of the internal portions of the operating system the user code interfaces with.

L

Lancaster Assembler

An Assembler for the 6502 and 65C02 written by Alan Phillips of the Department of Computing at Lancaster University in 1987. It was a 16K ROM image for the BBC Microcomputer and Acorn Electron which could be written into Sideways RAM or written to an EEPROM.

Laptop

A class of computer which has a built-in keyboard and screen – usually attached via a hinge so when closed both are covered.

Latch

A logic gate which retains its state once the source data has been removed. This is like a memory location, except that its state is being presented to external devices.

An example of this is the ROM paging register on the BBC Microcomputer. That register selects the active paged rom which can be accessed by the CPU.

When the processor writes a byte to that register, the value remains active even though the processor has moved on to other things.

LIFO Last In, First Out

An object like a stack which accepts data for later retrieval. When an item is removed, or popped, from the stack the last item pushed onto it is retrieved.

Linux

A Unix like operating system predominately for Intel and ARM based machines.

Little-Endian

The most common form of Endianess where the least significant byte is first, with each more significant byte following it until the most significant byte is reached.

Long

A long usually represents a double Word of memorys. On 8-bit machines a long usually represents 4 bytes or 32-bits. On modern machines a long usually represents 8 bytes or 64-bits.

Low

On an electrical bus a low signal usually designates a 0. That is, on a $5VTTL$ system, if a line is at $0V$ then it is usually indicating a 0. The Address Bus and Data Bus of a microcomputer uses this level of signalling.

Note that for active low buses, a low value represents a 0,

M

Machine Code

The language a CPU executes. This can be written in raw binary, bytes or more usually with an Assembler for that specific architecture.

Macintosh

The main successor line of machines after Apples 8-bit machines. The first machine was the Macintosh 128 which had a built-in monochrome monitor but introduced te GEM windowing system to the general public.

Later machines replaced GEM with MacOS and is the current desktop and laptoop machines procuded by Apple.

MacOS

The OS used on computers manufactured by Apple. Versions from 10 are used on the modern Intel and now ARM based Macintosh machines. Versions up to 9 were on the earlier machines.

Mainframe

A class of computer which is designed for computing large amounts of data. Mainframes are, historically, very large in size. The original machines were the size of a room. These days a small mainframe can be housed in a single rack.

Maskable interrupt

A form of hardware interrupt where the microprocessor can temporarily disable, hence mask, interrupts so that a time critical section of code can run without interruption.

This is usually complemented with NMI which is a type of interrupt which cannot be masked.

Matrix Brandy Basic

An Open Source version of BBC Basic, forked from Brandy Basic. It aims to improve BASIC VI compatibility to the original.

MCGA Multi-Color Graphics Adapter

A short lived standard which superseded EGA. It is similar to VGA in that it had a 256-color mode which was sometimes referred to as MCGA, and uses 15-pin analog connectors.

MCGA was short lived – the PS/2 Model 25 and Model 30 were discontinued by 1990, and the only manufacturer to produce a clone of this display adapter was Epson, in the Equity Ie and PSE-30. The PS/2 chipset's limited abilities prevent EGA and high-resolution multi-color VGA display modes.

MDA Monochrome Display Adapter

A monochrome analog video graphics standard introduced with the IBM PC in 1981 providing text resolution of 80 columns by 25 lines.

Although MDA could theoretically display graphics up to 720×350 pixels, it could not address individual pixels as it could only display characters from ROM so the only graphics were via ASCII art.

Microcomputer

A class of computer which followed the minicomputer. The primary feature of a microcomputer is that the CPU on a single chip. In the earlier classes of machine the CPU tended to be multiple components or even

boards.

Microcontroller

An all in one chip which has a CPU and GPIO lines to interface to the outside world. Some early microprocessors had some integrated IO, like the 6510 used in the C64 had a few lines.

Today, many microcontrollers are far more powerful. Some like the Raspberry PI Pico can emulate a system from the 1980's in real time.

Microdrive

A type of tape storage created by Sinclair originally for their Spectrum with the ZX Interface 1, and later with the Sinclair QL and clones like the BT Merlin Tonto.

Microprocessor

The CPU in a microcomputer.

Microsoft

A software giant who started as **Micro Soft** around 1975 with a version of BASIC now known as MSBasic for various systems of the era.

MIDI

A serial protocol for controlling musical instruments.

Minicomputer

A class of computer which preceded the microcomputer. The main difference is that in a minicomputer the CPU is usually an entire board within the machine using discrete components whilst a microcomputer has it's CPU on a single chip.

MiSTer

An emulation system which can emulate multiple machines at the hardware level. They can be 8-bit machines like the BBC Microcomputer or C64 to arcade cabinets.

Modern Computing

Anything that's too young to be Retro Computing.

MOS **Machine Operating System**

The name of the OS for the BBC Microcomputer range of machines.

MOS

The original manufacturer of the 6502 and a subsidary of Commodore.

Motherboard

The main board of a microcomputer which contained the core components including the CPU and RAM. For machines like the Apple II or the IBM PC which had cards which plugged into the machine, the motherboard refered to the board those cards plugged into.

Motorola

A manufacturer of microprocessors including 6809 and the 68000.

MSBasic

A version of BASIC written by Microsoft for 8-bit processors from around 1975.

MS-Dos

An OS from Microsoft for 8086 based systems. It had some heritage from CP/M.

MS-Windows

Initially a graphical interface layer over MS-Dos and then an OS in its own right produced by Microsoft.

N

National Semiconductor

A former manufacturer which produced the 32016 microprocessor. It is now part of TI.

Negative Flag

In Assembly Language the Negative flag is used to indicate that the most significant bit in the result of some operation was set. If that bit was set then the result would be negative in two's complement.

NetBSD

An OS based on BSD and Unix. *See* FreeBSD, OpenBSD

Nibble

4 bits of memory, or 2 nibbles form a byte. A nibble can be represented by a single hexadecimal digit.

NitrOS-9

A port of the OS-9 OS for the TRS-80 Color Computer.

NMI Non Maskable Interrupt

A form of hardware interrupt where the microprocessor has to service the interrupt – they cannot be disabled like Maskable interrupts.

NMOS n-Type Metal-Oxide Semiconductor

Uses n-type MOSFETs to implement logic gates and other digital circuits. It was effectively replaced by CMOS which proved more reliable and did not generate as much heat.

NMOS AND-by-default logic can produce unusual behaviour in NMOS components. The illegal opcodes in the original 6502 were caused by this behaviour – they did not occur with the CMOS 65C02.

O

Octal

The Base 8 number system which uses the decimal digits $0 \ldots 7$. It uses exactly 3 bits of storage. In this system, $7 + 1 = 10$.

One's complement

A numbering scheme that handles negative values where the most significant bit is set for negative values and reset for positive. Unlike two's complement, zero can be both positive and negative. *e.g.* -1 is represented as `0xFE` in a byte, whilst -0 is represented as `0xFF`.

To negate a number, you invert it: $neg(n) = \neg n$

see two's complement

Open Hardware

Hardware equivalent to Open Source.

Open Source

Software who's source is freely available and is redistributable under a license.

OpenBSD

An OS based on BSD and Unix. *See* FreeBSD, NetBSD

Opus DDOS

A double density disk filing system partially compatible with Acorn DFS.

OS **Operating System**

The core program of a Microcomputer which provided an API to the hardware of the system.

For many systems like the Spectrum this also contained the core language like BASIC with the hardware being accessed directly from it.‘

Others like the BBC Microcomputer had an OS to which languages and software would interact with called MOS.

Others like the Amstrad Word Processors used or were based on CP/M.

OS-9

OS-9 is a family of real-time, process-based, multitasking, multi-user operating systems, initially released in 197 for the Motorola 6809 microprocessor. Later versions supported the 68000, ARM and Intel processors. Today it is still used for embedded systems.

P

Page

A page represents an area of memory or of a file which can be accessed in a single block.

For the 6502 family, a page is a continuous block of 256 bytes of memory numbered by the upper 8-bits of the Address Bus. Hence, the 6502 stack is located in page `0x01` in memory which is the addresses `0x0100` …`0x01FF`.

Paged ROM

A ROM in a BBC Microcomputer which could be paged in when it was needed. Each ROM could be up to 16K as it was located in the addresses `0x8000` ...`0xBFFF`.

Paged ROMs could either be a language, a ROM FS image or a Service ROM. BBC Basic was itself a Paged ROM.

Panos

OS for the 32016 second processor, Cambridge Co-Processor, Acorn Business Computer and Cambridge Workstation.

Parallel

Data transfers occurring along several lines simultaneously.

PATA Parallel ATA

A parallel interface for connecting hard drives and cd drives to computers using a ribbon cable. It was originally called IDE.

PC Personal Computer

Originally this described a computer used by an individual but has since been used to refer to IBM computers and compatibles since the late 1980's.

PC-Dos

An OS from Digital Research for 8086 based systems. It had some heritage from CP/M. A version was available for the BBC Master 512.

PCMIA

A technical standard for an expansion card interface for Laptops and PDAs, first published in 1990. It initially supported a 16-bit bus, but later a 32-bit bus was developed. Newer ports supported both but older ports did not support 32-bit.

Some computer systems also used PCMIA. The Commodore Amiga A1200 and Amiga A600 had a 16-bit PCMIA slot to provide RAM, storage or networking capabilities to the machine – although RAM on this interface was very slow.

PDA Personal Digital Assistant

A small handheld device which would be used to manage the daily life of the user. It usualy containd a standard list of applications including

Address Book, Calendar, Calculator and Note pad.

Although popularity waned pretty quickly – mainly due to poor battery life for many systems, today everyone uses these same applications as they are a standard part of modern smart phones.

PDP-11

A model of computer by Digital.

Peripheral

Any device connected to the computer either externally or internally. Internal peripherals include analogue, joystick and printer ports *etc.* but components like RAM or ROM are not.

PETSCIIPET Standard Code of Information Interchange

The character set used on Commodore 8-bit machines, starting with the CBM PET.

PITube

A modern interface which can connect a Raspberry PI to the Tube interface on the BBC Microcomputer or Acorn Electron. The Raspberry PI runs the PITube software instead of an OS so that an Second Processor would be emulated. It emulates all the original processors but also some additional ones like the PDP-11.

Later releases also allow the HDMI display on the Raspberry PI to be used as a second display.

Plus 1

For the Acorn Electron, an expansion which provided a parallel printer port, an analogue input port and two cartridge slots.

Plus 3

For the Acorn Electron, an expansion which provided a 3.5" floppy drive. The interface attached to the back of the Acorn Electron and, due to the higher powe load, replaced the original power supply with a more powerful one. Other interfaces like the Plus 1 would be attached on the back of the Plus 3.

POSIX Portable Operating System Interface

A set of standards first published in 1988 for maintaining compatibility between operating systems.

PowerPC Performance Optimization With Enhanced RISC – Performance Computing

A processor architecture created in 1991 by AIM introduced in 1992. Sometimes abbreviated as PPC.

Apples used it for their desktop and laptop lines from 1994 to 2006. It was also used the Curiosity and Perseverance rovers on Mars and a variety of satellites Today it's used in embedded systems and AmigaOS 4.

Printer

A device which would print text onto paper.

Prism VTX5000

A 12/75 modem for the 48K ZX-Spectrum which allowed access to Prestel and other Viewdata systems.

PC Program Counter description

PROM Programable Read Only Memory

A type of ROM which can be programmed only once.

Pyromaniac riscos Pyromaniac

An alternative implementation of RISC-OS for non-ARM systems.

R

RAM Random Access Memory

A chip which contains memory that can be read and written to. The contents of the memory is lost when the power is removed.

RAM Disk

An area of RAM which is used as if it was an external drive. These were usually available on machines with plenty of memory like Linux, however

some early machines like the Spectrum 128K had them. As a RAM Disk is stored in RAM it's contents is lost on reboot or power off. However, due to them being in RAM they are very fast.

Raspberry PI

A type of Single Board Computer computer which uses an ARM processor. It's model names were initially based on those of the BBC Microcomputer as its designers grew up with those machines.

Raspberry PI Pico

A microcontroller which has two ARM cores. Some microcomputer systems have been emulated on this device.

Register

A variable internal to a CPU.

Research Machines

A British computer manufacturer, founded in 1973 which produced many microcomputers during the 1970's and 1980s including the RM 380Z.

Retro Computing

A modern day hobby using, repairing or emulating computers at least 30 years old or more. The actual age is up for debate, the figure I present here is a rough guess. *e. g.* the original Sony Playstation is now considered retro by some.

Reuters

A british news agency formed in 1851. They used a custom version of the BBC Model B known as the Reuters APM.

Reuters APM

A BBC Model Bs in a Eurocard format, commissioned by Reuters.

Reuters OS

A version of MOS which ran on the Reuters APMs.

RF Radio Frequency

A signal, either audio or video carried over radio. For computer displays, RF was either in the VHF or UHF frequency ranges and carried the video display. As this was an analogue signal, the quality was not sufficient for

high resolution displays so the majority of displays were 40 characters horizontally at a maximum.

80 characters were possible but were not easily read – those video modes required a monitor to be usable.

RGB **Red Green Blue**

The three primary colours of phosphor used on CRT display tubes. Most computers which have colour displays use the RGB colour space for generating colours. In its simplest binary form, RGB can provide 7 colours. Table 2 on page 1107 shows the RGB colour set.

Table 2: RGB colours

Dec	BGR	Colour	
0	000	Black	No colour
1	001	Red	
2	010	Green	
3	011	Yellow	Red + Green
4	100	Blue	
5	101	Magenta	Blue + Red
6	110	Cyan	Blue + Green
7	111	White	Red + Green + Blue

Red is usually Bit-0 hence shown in as BGR above

RGBI

A type of RGB monitor which used four signals instead of three – the second signal being the intensity or brightness. This expanded the RGB range to 16 colours. Table 1 on page 1087 shows the 16 colours used by CGA.

Colour 6 would normally encode to being a dark yellow but most RGBI monitors would follow the CGA standard and adjust the hue from dark yellow to a brown by reducing the green level by $\frac{1}{3}$.

Ribbon

A cable consisting of multiple wires which are attached together. A cable is usually used with IDC connectors as those connectors can be simply crimped to a ribbon.

Hard drives using the IDE or PATA buses as well as floppy drive used

ribbon cables when installed internally.

RISC Reduced Instruction Set Computer

A type of instruction set for CPUs which have a minimum number of instructions, enough to perform tasks. It evolved from CISC architectures as an effort to simplify chip design, especially at a time when the process was done manually.

ARM processors are a type of RISC.

RISC-OS

An OS for the original Acorn ARM processors which was loosely based on MOS and released in April 1989 as RISC-OS 2 – it was originally named Arthur. Originally for the Archimedes, it is still available today for the Raspberry PI.

See Pyromaniac for an alternate, non-ARM implementation.

RM 380

A predecessor to the RM 380Z which did not have a disk dive. I did have one of these during the 1990's. Unlike it's successor whose case was black, this had a white and blue case and only had a cassette interface.

RM 380Z

A Z80 based microcomputer by Research Machines which ran CP/M.

RM 480Z

A variant of the RM 380Z which was in the all-in-one style with keyboard and machine in one. This was used in UK schools – I remember my school had an entire room of these which was rarely used.

RM Nimbus

A range of personal computers by Research Machines sold from 1985 until the early 1990's, based around the IBM PC but not entirely compatible with it.

Rollover

For keyboards, rollover is the number of concurrent keys that can be pressed at the same time. This usually applies to keys other than the shift or control keys.

e.g. "2 key rollover" means that a system can detect 2 keys being pressed

at one time. If more than 2 are pressed then either only the first two or last two pressed are detected – dependent on the implementation.

ROM Read Only Memory

A chip which contains memory that can only be read to.

ROM FS ROM FileSystem

A filesystem on the Acorn Electron and BBC Microcomputer held on a Paged ROM. It allowed for games to be stored on a cartridge with few changes as it shared the same format used by cassettes.

Rotate

A rotate is like a shift where a value is shifted one bit to the left or right. The difference is that the bit being shifted in comes from a flag, usually the Carry Flag.

This allows for rotates comprising multiple bytes or values to be performed using multiple instructions.

see shift

RS-232

A serial communication wiring standard used to connect a computer to devices or other computers. Each end is usually either a 9-pin or 25-pin dsub connector, but not always.

The full RS-232 standard consists of two serial channels where the second channel was used usually for printers, but it was rarely provided for by the 1980's.

RS-232 is electrically compatible with RS-423.

RS-232C

A compact version of RS-232 using just the 9-pin dsub connector.

RS-423

A serial communication wiring standard used to connect a computer to devices or other computers. Similar to and compatible with RS-232, this is a 5-wire system used by the BBC Microcomputer family.

RTS **Request To Send**

A line on a RS-232 or RS-423 serial cable which the sender asks the receiver if it can send data. The opposite end for this line is CTS.

RunCPM

A Z80 CP/M emulator.

S

SAS

A type of interface for connecting hard drives and cd drives to computers. It is the modern equivalent to SCSI.

SATA **Serial ATA**

A type of interface for connecting hard drives and cd drives to computers. It is the modern equivalent to PATA and IDE.

Scanner

A device which would take an image of a page and convert it into an image.

SCSI

A communications protocol for connecting multiple devices to a computer. These devices can be anything, ranging from hard drives, cd drive to scanners.

The original bus was parallel and could handle up to 15 devices. It had to be terminated with a resistor pack at both ends.

Modern SCSI is serial based and has evolved into SAS and the protocol used in USB.

SDL **Simple DirectMedia Layer**

A cross-platform software development library that provides a hardware abstraction layer for multimedia hardware components. It allows developers to create applications that can run on various operating systems without needing to manage the specific details of each platform.

Many emulators and languages like BBC Basic use SDL to enable them

to operate across multiple environments.

Second Processor

The BBC Microcomputer supported the use of a second CPU attached to the main computer over the Tube interface. When in use, the Second Processor ran the applications leaving the original CPU to manage the keyboard, display and peripherals.

See Tube, PITube

Sega

A Japanese video game company.

Sega Master System

A 8-bit game console by Sega. First released in 1985 it was produced until the early 1990's – except for Brazil where it is apparently still under production. Table 3 shows the lifetime of this console in various parts of the world.

Table 3: Lifespan of the Sega Master System

Territory	Released	Discontinued
Japan	1985	1991
North America	1986	1992
Europe	1987	1996
Korea	1988	1994
Brazil	1989	Present

Serial

Data transmitted along a single line can only be sent one bit at a time, hence is sent serially.

Serial devices on microcomputer's usually use RS-232 or RS-423 as their cabling types, although modern interfaces like SATA and USB are also serial.

SerialBus

A serial bus on the C64 and related machines to connect them to printers and disk drives.

Service ROM

A ROM which provides a service. On the BBC Microcomputer, this would be a File System or the driver for some hardware.

Shift

A shift is where a value is shifted one bit to the left or right. For left shifts, the least significant bit is reset and effectively multiplies the value by two.

For right shifts either the most significant bit is reset or it is kept with it's original value, even though that bit was shifted right. The latter enables a values sign, when two's complement is used.

see rotate

Sideways RAM

A 16K block of memory located in the address space that a Paged ROM on a BBC Microcomputer. This ram could be used to hold Paged ROM images or as additional memory buffers. The BBC Master 128 had 64K of Sideways Ram in four 16K blocks, whilst the Acorn Electron had cartridges which held two 16K blocks which had a battery backup to keep the ram contents when the computer was turned off.

Sinclair

Manufacturer of the popular Spectrum, it's predecessors the ZX-80 and ZX-81 as well as the Sinclair QL. It was later purchased by Amstrad.

Sinclair Basic

The flavour of BASIC for the Spectrum by Sinclair, although it first appeared on the ZX-80 and ZX-81.

Sinclair QL

Sinclair's attempt at a successor to the Spectrum but it didn't take off. It did have some clones like the BT Merlin Tonto.

QL = Quantum Leap

Single Board Computer

A computer which is on a single board. Although today this name targets machines like the Raspberry PI, many of the computers from the 1980's were also only a single board.

Solidisk DFS Solidisk Disk File System

A disk filing system partially compatible with Acorn DFS.

Spectrum

Sinclair's most popular machine which succeeded the ZX-81. Initially it was available as a 16K and 48K machine with a rubber-key keyboard. Later models had a slightly better keyboard with the 48K+ and the original 128K machine.

When Amstrad purchased Sinclair, the +2 and +3 machines had a proper keyboard.

Spectrum +2

An update to the Spectrum 128K which had a proper keyboard and cassette deck built into the machine. Its case was similar in design to the CPC464.

Although you could load from an external cassette deck, you could not save to one without modifying the motherboard.

Spectrum +2A

An update to the Spectrum +2 which had the same motherboard as the Spectrum +3. The only difference was that the components to support the floppy drive not present. In theory support for a floppy drive could be provided simply by installing the missing components. The ROM was identical and determined which model the machine was simply by detecting the presence of the floppy controller.

Spectrum +3

An update to the Spectrum 128K which had a proper keyboard and cassette deck. Its case was similar in design to the CPC464.

Spectrum 128K

The successor to the Sinclair's Spectrums which had 128K of RAM and AY-3-8912 sound chip.

Spectrum+

An update to the Spectrum 48K which had a keyboard and case similar to the Spectrum 128K.

Spreadsheet

An application which manages a grid of cells into which text, numbers or formulae could be entered. If a value changes then any formulae which referenced that value is updated.

Springboard

A coprocessor card for the IBM PC produced by Acorn which provided an ARM coprocessor and either 1 or 4 of memory. Communication between the PC and the ARM processors was done over the Tube originally produced for the BBC Microcomputer.

Stack

A Last In First Out (or LIFO) block of memory where data is inserted and then removed at a later stage. Every microprocessor has a stack which it uses to store the return address when it makes a call to a Subprogram. It is also used by Machine Code to store the processors registers during subroutines and interrupt handlers.

Static RAM

A form of RAM which is simpler than DRAM as it does not need regular refreshing. It also tends to be faster. The downside is that Static RAM uses a lot more transistors than DRAM, so chip capacities is less based on the physical size of the chip.

However, registers within a processor are usually in the form of Static RAM due to their higher speed.

STM32

A family of 32-bit ARM microcontrollers.

Subprogram

In TI Basic, a predefined general-purpose procedure accessible to the user through the **CALL** statement. Subprograms extend the capability of TI Basic and are usually not easily programmed themselves in TI Basic.

Devices like the TI Speech Synthesizer were usable using this mechanism.

System DFS

A disk filing system for Acorns Acorn System range of computers and the Acorn Atom. It was the predecessor for Acorn DFS used on the BBC Microcomputer.

T

Tandy **Tandy Corporation**

A manufacturer who produced the TRS-80 family of microcomputers.

Tape deck

see cassette deck

TCP/IP

A networking protocol which forms the basis of how the Internet works.

Teensy

A family of ARM microcontrollers

TI **Texas Instruments**

A major electronics manufacturer who produced minicomputers until the early 1980's. Between 1979 and 1983 they had a short-lived microcomputer called the TI99/4A.

TI Basic

A version of BASIC for the TI99/4A microcomputer.

TI Calculator

A range of advanced calculators by TI which can run programs and systems. Some like the TI-83+ and TI-84+ can even run BBC Basic.

TI Speech Synthesizer

An expansion module for the TI99/4A which provided the computer with a Speech Synthesizer.

TI99/4A

A microcomputer from TI which was based on its minicomputer range.

TI99/4

The original version of the TI99/4A. The main difference was that this used a simpler chicklet keyboard whilst the newer TI99/4A had proper keys.

Torch Computers

A manufacturer of hardware for many microcomputers if the 1980's.

TRS-80

A microcomputer by Tandy which used the Z80 processor. Colloquially known as the "Trash-80". One of the 1977 Trinity if machines.

TRS-80 Color Computer

A all-in-one microcomputer by Tandy. Released in September 1980 it was discontinued in 1991.

Despite its name, it did not use the Z80 processor of the TRS-80 but the Motorola 6809 so was incompatible. However, this did allow OS-9 to be ported to this machine – known as NitrOS-9.

TTL Transistor Transistor Logic

A form of chip logic which uses 5*V* logic levels on the outputs. More recent devices use 3V3 levels and cannot be used with TTL devices without being buffered.

Tube

An interface on the BBC Microcomputer, Acorn Electron and Acorn Atom which allowed the use of a Second Processor. The BBC Master 128 had a second internal Tube interface – although only one of the two could be used at the same time.

The Springboard was a IBM PC card also used the Tube.

See PITube

TV Television

A form of display used primarily for home entertainment purposes which received a video signal over RF. They were used in the early home computing era as the primary display for computers as the user would have one rather than having a separate more expensive monitor. At that time they were all based on CRT tubes.

Two's complement

A numbering scheme that handles negative values where the most significant bit is set for negative values and reset for positive. Zero is positive. *e.g.* −1 is represented as `0xFF` in a byte.

To negate a number: $neg(n) = 0 - n$

see one's complement

U

UHF **Ultra High Frequency**

A band of radio frequencies between $300MHz \ldots 3GHz$.

For TV this carried channels $14 \ldots 69$ between the frequencies $470 \ldots 806MHz$. UHF was more commonly used for TV during the early home computer era, with channel 32 being the most common one used

ULA **Uncommitted Logic Array**

A chip which has been specially created for a specific purpose. Many manufacturers designed a custom chip to handle certain features like the display and used a ULA to implement this chip.

Unix

An OS which originated from the 1970's and evolved into many modern day OSs like FreeBSD, NetBSD and OpenBSD. Some OSs like Linux are based on unix, although they have no direct relation to it.

Unix-Time

A clock with an epoch of 1970.0 and a resolution of 1 second. Specifically, it counts the number of seconds since Midnight on 1st January 1970. Many operating systems including Linux, FreeBSD *e.g.* use this system.

Originally this was a 32-bit signed integer so it could represent a time ±68 years from 1970. However, like Y2K this will cause an issue in 2038 with what is known as Y2K38 where this clock will roll over and the clock will suddenly show time since 1902.

To fix this, modern Unix-Time clocks are 64-bit moving the rollover date from 2038 to 292 billion years in the future.

USB **Universal Serial Bus**

A serial bus used in modern machines to attach a variety of devices. Internally it uses a version of the SCSI protocol.

User land

The part of an OS outside of the kernel, which can be directly accessed by the user.

U.V.

Ultra Violet light. Some types of semiconductors like EPROM, can be erased by being placed under a U.V. light source.

V

VDU

On the BBC Microcomputer this represents the character set. Based on ASCII for values above 31, it has a suite of commands defined for codes less than 32 that differ to ASCII. These commands range from changing the screen mode to drawing graphics in graphical modes.

VGA **Video Graphics Array**

An analog video graphics standard introduced with the IBM PC in 1987 providing a $640x480$ pixel resolution. Unlike the earlier MDA, CGA and EGA standards which used a 9-pin dsub connector, VGA uses a 15-pin dsub connector.

Since the original specification was first released, the standard has been updated to handle resolutions up to 2048×1536 pixels and the cable is still commonly available even though newer digital standards are now available.

VHF **Very High Frequency**

A band of radio frequencies between $30 \ldots 300MHz$.

For TV this carried channels $2 \ldots 13$ between the frequencies $54 \ldots 216MHz$. UHF was more commonly used for TV during the early home computer era.

Vic20 **Commodore VIC-20**

An early 8-bit machine by Commodore which came out after the CBM PET and before the C64. It had many of the features of the C64 except it had minimal RAM and early version of the chips in the C64.

View

A Word Processor for the BBC Microcomputer. It was available as a cartridge for the Acorn Electron and as standard for the BBC Master 128.

Viewdata

A type of bulletin board system accessed over a modem, utilising Teletext graphics so was restricted to 40*x*25 character screen.

ViewSheet

A Spreadsheet for the BBC Microcomputer. It was available as a cartridge for the Acorn Electron and as standard for the BBC Master 128.

V-Sync Vertical Sync

A marker at the start of a frame of video.

On most multiple-line video signals it is a line separate from the actual video signal. *e.g.* RGB and VGA cables have a separate vertical sync line.

On single line signals like Composite Video or RF this is a pulse at the start of the frame used for the same purpose.

In software, this can be an event which occurs when the signal occurs allowing software to synchronize themselves to the actual screen generation.

W

Watford 62DFS

Identical to Watford DFS except it has a second catalogue doubling the number of files per disk from 31 to 62.

Watford DDFS

A double density disk filing system partially compatible with Acorn DFS.

Watford DFS Watford Disk File System

A disk filing system partially compatible with Acorn DFS.

Wi-Fi™

A family of network protocols used over a radio connection.

WIMP Windows, Icons, Menus and Pointer

A graphical user interface like GEM, XWindows or MS-Windows.

Word of memory

A word is a block of bits representing a value. On 8-bit machines a word usually represents 2 bytes or 16-bits. On modern machines a word usually represents 4 bytes or 32-bits.

On some older systems, the terms word and byte were interchangeable.

Word Processor

An application to which words were processed into a document. Word-processors of today are not the same as they do so much more than what the original wordprocessor could do. Back in the day, a wordprocessor just handled words and not much more.

X

Xenix

A Unix System V based operating system by Microsoft released between 1980 and 1985.

XWindows

A graphical user interface for Unix style OSs like Linux.

Y

Y2K Year 2000 problem

The year 2000CE, or more commonly the Year 200 bug, where many clocks which used 2-digit years had a problem because they could only reference the 20th century with 19xx for the year.

Many retro computers and software suffer from this issue, although some have had a "fix" to solve the issue.

Y2K38 Year 2038 problem

The year 2038CE, sometimes called the Epochalypse, is a problem where the clock using Unix-Time rolls over from a positive 32-bit number to a negative one. This will happen on 19 January 2038 at 03:14:08. 32-bit systems will then show the next second as being on 13 December 1901.

Modern systems have moved from a 32-bit signed integer to a 64-bit signed integer which moves the rollover time to 292 billion years in the future.

Z

Z80

A popular 8-bit microprocessor from the late 1970's which only ceased being manufactured in its original form dueing the early 2020's . Many microcomputers of the era were based on this processor including the entire Sinclair and Amstrad ranges.

The popular CP/M operating system ran on this processor.

Today it is available in a modern format and is popular on high-end calculators.

Although manufactured by Zilog, the Z80 was designed as a replacement for Intels 8080 processor.

[2] *Z80 CPU User Manual*, (Zilog Inc Aug. 2016)

Z80-OS

A mini operating system by Acorn which operates with the Z80 second processor on the glsbbcmicro. It acts a go-between with Acorns CP/M.

Zero Flag

In Assembly Language the Zero flag is used to indicate the result of some operation was zero.

Zero-page

On the 6502 family of processors, page 0 containing addresses `0x0000...00FF` is known as the zero-page. There are several address modes specific to this page allowing for instructions which uses this page of memory to

be one byte shorter. Not only does this save in the amount of memory used by a program, it runs faster as it removes a cycle needed to fetch that byte from memory.

Zilog

The original manufacturer of the Z80.

ZX Interface 1

An interface for the original 48K ZX-Spectrum which enabled the use of MicroDrive cartridges and provided both a RS232 port and a rudimentary network interface.

ZX-80

The first home computer produced by Sinclair. It was initially made available as a kit for hobbyists to build, then in pre-built machines. The ZX-80's display was generated by the CPU rather than dedicated hardware.

ZX-81

The second home computer produced by Sinclair. It followed the ZX-80 on which it was closely based upon, only differing by a ROM that was twice the size and a 'fast' mode which sped up the computer by turning off the display. Like their ZX-80, the ZX-81's display was generated by the CPU rather than dedicated hardware. It was replaced by the extremely popular ZX-Spectrum.

ZX-Printer

A small printer for the ZX-81 and ZX-Spectrum which used metalic paper rolls about the same size as a Till Receipt printer. The printer "printed" onto the paper by burning off the top aluminized layer leaving a blackened spot where a pixel would be located. I remember it smelt whenever it was used and the paper was expensive and hard to get at the time.

Basic

Legend: Page, **Definition**, **Glossary**, **Errata**, *see alternate entry*.

D

data 11

E

else 9

G

gosub 10
goto 10

I

if 9

L

list 6
load 7

R

read 11
return 10
run 6

S

save 7

T

then 9

BBC Basic

Legend: Page, **Definition**, **Glossary**, **Errata**, *see alternate entry*.

A

abs 20
Acorn Electron 13
acs 20
ADVAL
    (0) 638
    (n) 639
adval 21
auto 15

B

BBC Microcomputer 13

D

delete 16

E

else *see* if

I

if 19
INKEY
    -256 643
    key code 642
    timeout 641

R

renumber 17

T

then *see* if

Sinclair Basic

Legend: Page, **Definition**, **Glossary**, **Errata**, *see alternate entry*.

D

data 26

F

for **27**

G

go sub 29
go to 30

I

if 28

L

let 31

N

next *see* for

R

read 32
restore 33
return 34

S

step *see* for

T

then *see* if

TI Basic

Legend: Page, **Definition**, **Glossary**, **Errata**, *see alternate entry*.

C

CALL 51
- CHAR **51**
- COLOR **52**
- GCHAR **53**
- HCHAR **55**
- SCREEN **55**
- SOUND **56**
- VCHAR **59**

CALL CLEAR **51**

D

DATA 38
DISPLAY 38

E

ELSE *see* IF
END 39

F

FOR **40**

G

GO TO 41
GOTO 41

I

IF 42
INPUT 44

L

LET 42

M

Musical Frequencies **58**

N

NEXT 43
- *see* FOR

O

ON GO TO 43
ON GOTO 43

P

PRINT 45

R

READ 48

REM 48
RESTORE 48
STOP 49
Subprogram *see* CALL

S

STEP *see* FOR

T

THEN *see* IF

6502 Microprocessor Family

Legend: Page, **Definition**, **Glossary**, **Errata**, *see alternate entry*.

A

ADC 107, **134**
AND **136**
ASL 98, **137**

B

BCC 82, **138**
BCS 82, **139**
BEQ 82, **140**
BIT 114
BMI 82, **143**
BNE 82, **144**
BPL 82, **145**
BRA 80, 116, **146**
BRK **147**
BRL 81, **150**
BVC 82, **151**
BVS 82, **152**

C

CLC 107, **153**
CLD **154**
CLI **155**
CLV **156**
CMP **157**
COP **158**
CPX **160**
CPY **161**

D

DEC 103, **162**
DEX 103, **163**
DEY 103, **164**

E

EOR 93, **165**

F

Flags 77

I

INC 103, **167**
INX 103, **168**
INY 103, **169**

J

JML *see* JMP
JMP 81, 116, **170**
  bug **171**
JSL *see* JSR
JSR 85, **172**, 204
  conditional 88

L

LDA **174**
LDX **175**
LDY **176**
LSR **177**

M

MVN **178**
MVP **179**

N

NOP **180**

O

ORA **181**

P

PC 79
PEA **182**
PEI **183**
PER **184**
PHA **185**
PHB **186**
PHD **187**
PHK **188**
PHP **189**
PHX **190**
PHY **191**
PLA **192**
PLB **193**
PLD **194**
PLP **195**
PLX **197**
PLY **198**

R

REP **199**
ROL 98, **200**
ROR 99, **201**
RTI **202**
RTL 86, 203, **204**
RTS 86, 203, **205**
  conditional 89

S

SBC 112, **206**
SEC 112, **208**
SED **209**
SEI **210**
SEP **211**
STA 116, **212**
STP **213**
STX **214**
STY **215**
STZ 116, **216**
SWA *see* XBA

T

TAD *see* TCD
TAS *see* TCS
TAX **217**
TAY **219**
TCD **220**
TCS **221**, 230
TDA *see* TDC
TDC **222**
TRB 114, 142, **223**
TSA *see* TSC

TSB 114, 142, **225**
TSC **226**
TSX **228**
TXA **229**
TXS 221, **230**
TXY **231**
TYA **232**
TYX **233**

W

WAI **234**
WDM **235**

X

XBA **236**
XCE **237**

Z80 Microprocessor

Legend: Page, **Definition**, **Glossary**, **Errata**, *see alternate entry*.

A

A **242**
A' **242**
ADC 108, 256, **276**
ADD 108, 256, **278**
AND 260, **280**

B

B **243**, 244
BC **243**, 244
BIT 114, **281**

C

C **243**, 244
CALL 82, 85, 87, **283**
CCF **285**
CP 257, **286**
CPD 244, **288**
CPDR 244, **290**
CPI 244, **291**
CPIR 244, **292**
CPL 95, 255, **293**

D

D **243**, 245
DAA 264, **294**
DE **243**, 245
DEC 104, 255, **296**
DI **298**
DJNZ 83, 244, **299**

E

E **243**, 245
EI **301**
EX **302**
EXX **304**

F

F **242**
F' **242**
Flags 77

H

H **243**, 245
HALT **305**
HL **243**, 245

I

I **247**
IM **306**
IN 244, **308**
IN F,(C) **309**
INC 104, 255, **310**
IND 244, **312**
INDR 244, **314**

INI 244, **315**
INIR 244, **316**
IX **246**
IY **246**

J

JP 81, **317**
JR 80, **319**
JSL 84
JSR 84

L

L **243**, 245
LD **320**
    LD r,(HL) **321**
LDD 244, **330**
LDDR 244, **332**
LDI 244, **333**
LDIR 244, **334**

N

NEG 95, 255, **335**
NOP **336**

O

OR 260, **337**
OTDR 244, **339**
OTIR 244, **340**
OUT 244, **341**
OUTD 244, **343**
OUTDR *see* OTDR
OUTI 244, **345**
OUTIR *see* OTIR

P

PC 79, **245**
POP 246, **346**
PUSH 246, **348**

R

R **247**
Register
    A **242**
    A' **242**
    B **243**, 244
    BC **243**, 244
    C **243**, 244
    D **243**, 245
    DE **243**, 245
    E **243**, 245
    F **242**
    F' **242**
    H **243**, 245
    HL **243**, 245
    I **247**
    IX **246**
    IY **246**
    L **243**, 245
    PC **245**
    R **247**
    SP **246**
RES **350**
RET 82, 86, **354**
RETI **355**
RETN **356**
RL 261, **357**
RLA 261, **359**
RLC 261, **360**, **363**
RLCA 261, **362**
RLD 264
RR 262, **364**
RRA 262, **366**

RRC 99, 262, **368**, **370**
RRCA 99, 262, **367**
RRD 264
RST 85, **371**
RTL 84

S

SBC 113, 256, **375**
SCF **377**
SET **378**
SLA 261, **382**
SP **246**
SRA 262, **384**
SRL **386**
SUB 113, 256, **388**

X

XOR 93, 260, **390**

BBC Basic Assembler Index

Legend: Page, **Definition**, **Glossary**, **Errata**, *see alternate entry*.

Symbols

[ ] 397, **405**

A

ASL 400

E

EQUB **405**
EQUD **406**
EQUS **406**
EQUW **405**

J

JMP 399

O

O% **401**
OPT 403, **406**

P

P% **401**

BeebASM Assembler Index

Legend: Page, **Definition**, **Glossary**, **Errata**, *see alternate entry*.

C

Command line
- -D **411**
- -boot **411**
- -d **411**
- -di **411**
- -do **411**
- -i **411**
- -o **411**
- -opt **411**
- -title **411**
- -v **411**
- -vc **411**
- -w **411**

Commodore C64 409
Constants
- CPU **438**
- FALSE **438**
- P% **438**
- PI **438**
- TRUE **438**

D

Directives
- ALIGN **421**
- ASM **421**
- ASSERT **421**
- CLEAR **422**
- COPYBLOCK **422**
- CPU **422**
- EQUB **423**
- EQUD **423**
- EQUS **423**
- EQUW **423**
- ERROR **424**
- FOR **424**
- GUARD **426**
- IF **427**
- INCBIN **429**
- INCLUDE **429**
- MACRO **429**
- MAPCHAR **432**
- ORG **433**
- PRINT **433**
- PUTBASIC **434**
- PUTFILE **434**
- PUTTEXT **435**
- RANDOMIZE **435**
- SAVE **436**
- SKIP **437**
- SKIPTO **437**
- SOURCELINE **437**

F

Functions
- <val **418**
- >val **418**
- ABS **439**
- ACS **439**
- ASC **439**
- ASN **439**
- ATN **439**
- CHR$ **440**

COS 440
DEG 440
EVAL 440
EXP 440
HI(val) *see* >val
INT 441
LEFT$ 441
LEN 441
LN 441
LO(val) *see* <val
LOG 442
LOWER$ 441
MID$ 442
NOT 442
RAD 442
RIGHT$ 443
RND 443
SGN 443
SIN 443
SQR 444
STR$ 444
STR$~ 444
STRING$ 444
TAN 445
TIME$ 445
UPPER$ 445
VAL 445

L

Labels
{} *see* =413
.*label 414
.^label 415
Local 413

O

Operators 417
AND 417
« 417
» 417
DIV 417
EOR 417
MOD 417
OR 417

V

Variables
CALLSTACK$ 433
FILELINE$ 433

Lancaster Assembler Index

Legend: Page, **Definition**, **Glossary**, **Errata**, *see alternate entry*.

Symbols

*ASSEMBLE **452**
*BASIC 449
*CLI **449**
*CONFIGURE
    LANG 449
*MODE **449**
*VIEW 449
*WORDWISE 449
$CLST **525**
$DEFCLST **525**
$DEFLST **525**
$FLEVEL **525**
$FS **525**
$LINE **525**
$LST **525**
$MC **525**
$MLEVEL **525**
$OBJECT **525**
$OS **525**
$OSHWM **525**
$PLENGTH **525**
$PRINTING **526**
$PWIDTH **525**
$SOURCE **526**
$TIME **526**
$TTL **526**
$VERSION **526**

A

ADE Assembler 453
AGO 509
AIF 509
ALEN 511, **513**
ASC **483**, 485
ASET 511, **512**, 513
ASTR 511, **514**
ASTRIP 511, **514**

C

CASC **484**
CHN 493
CHR 475
CLST 472
CODE **486**
CSTR **484**

D

DB **480**
DDB **480**
DEFPARS **522**
DEND 489
DFB **480**, 482, 485
DFDB **480**, **485**
DFW **480**
Directives
    AGO 529
    AIF 529
    AIFDEF 529
    AIFNDEF 530
    ALEN 530

ASC 530
ASET 530
ASTR 531
ASTRIP 531
CASC 531
CHN 532
CHR 532
CLST 533
CODE 533
CSTR 534
DB 534
DDB 534
DEFPARS 535
DEND 535
DFB 535
DFDB 536
DFS 536
DFW 536
DISP 536
DISP1 536
DISP2 537
DO 537
DS 537
DSECT 538
DW 538
ELSE 539
ENDM 539
EQU 539
EXEC 540
FI 540
FIN 540
HEX 541
IF 541
IFDEF 541
IFNDEF 542
INCLUDE 542
INFO 542
LFCOND 543
LOAD 543
LST 544
MACRO 545
MEXIT 545
MLEN 545
MSET 546
MSTR 546
MSTRIP 546
MSW 546
ORG 547
PAGE 547
QUERY 548
REP 548
SFCOND 548
SKP 549
STOP 549
STR 549
SYSCLI 550
SYSVDU 551
SYSVDU1 551
SYSVDU2 551
TABS 552
TIME 553
TTL 552
WAIT 553
WAIT1 553
WAIT2 553
DISP 478, 499
DISP1 499
DISP2 499
DO 496
ELSE 496
FIN 496
DS 490
DSECT 489
DW **480**

E

ELSE 495
EQU 489

F

FI 495
FIN 496

H

HEX **485**

I

IF 495
    ELSE 495
    FI 495
IFDEF 497
IFNDEF 497
INCLUDE 494
INFO 499

L

LFCOND 497
LST 469

M

MACRO 503
    ENDM 503
MEXIT **523**
MLEN 511, **513**
MSET 511, **512**, 513
MSTR 511, **514**
MSTRIP 511, **514**
MSW 492

O

ORG **487**

P

PAGE 473

Q

QUERY 500

R

REP 475

S

SFCOND 497
SKP 475
STOP 502
STR **483**
SYSVDU 477
SYSVDU1 477
SYSVDU2 477

T

TABS 471
TIME 474
TTL 474

V

View 451

W

WAIT 500
WAIT1 500
WAIT2 500
WordWise 451

Errata

Legend: Page, **Definition**, **Glossary**, **Errata**, *see alternate entry*.

B

Buffer Overrun **769**

I

INKEY
Acorn Electron **592**

K

Keyboard
Soft key pointer **729**

O

OSBYTE
`0x15` Flush specific buffer **622**
`0x80` Read ADC channel/buffer status **640**
`0x8A` Insert into buffer **648**
`0x91` Get character from buffer **653**
9F Write to speech processor@`0x9F` Write to speech processor **667**
`0xA0` Read VDU variable **668**
`0xA3` Enable or Disable printer and ADC **671**
`0xB3` Read Write primary OSHWM **710**
`0xFF` Startup Options **748**
BBC Master Compact **749**
BBC Master 128 **749**
OSWORD
`0x05` Read byte of I/O processor memory **758**
`0x0E` Read real time clock **769**
`0x0F` Write real time clock **772**

S

Sound
Semaphore **728**

V

VDU
19 **992**
20 **992**

Z

Z80
LD r,(HL) **321**

Alphabetical Index

Legend: Page, **Definition**, **Glossary**, **Errata**, *see alternate entry*.

Symbols

*. *see* *CAT
*/ *see* *RUN
*ACCESS *see* Acorn DFS
see Sprow LANMANFS
*ASSEMBLE
see Lancaster Assembler
*BACKUP *see* Acorn DFS
*BASIC 601
*BUILD *see* Acorn DFS
*BYE *see* Sprow *BYE
*CAT 601, **784**, 793
see Acorn DFS
*CDIR
see Sprow LANMANFS
*CLI *see* Lancaster Assembler
*CO. *see* *CODE
*CODE 601, 647, **777**
*COMPACT *see* Acorn DFS
*CONF. *see* *CONFIGURE
*CONFIGURE 571
BAUD 575
BOOT 575
CAPS 576
DATA 576
DELAY 577
DIR 577
DNS **816**
EMADDR **816**
EMADVERTISE **815**
EMLINK **814**
FDRIVE 577
FILE 578
FLOPPY 578
FS 573
GATEWAY **816**
HARD 578
IGNORE 578
LANG **579**
LOUD 580
MODE 579
PRINT 579
PS 573
QUIET 580
REPEAT 580
SCROLL 580
SPACE 574
TUBE 581
VIEW 581
*COPY *see* Acorn DFS
see Sprow LANMANFS
*DELETE *see* Acorn DFS
*DESTROY *see* Acorn DFS
see Sprow LANMANFS
*DIR *see* Acorn DFS
see Sprow LANMANFS
*DISMOUNT
see Sprow LANMANFS
*DRIVE *see* Acorn DFS
see Sprow LANMANFS
*DUMP *see* Acorn DFS
*EMINFO
see Sprow LANMANFS
*EMTEST
see Sprow LANMANFS
*ENABLE *see* Acorn DFS
see Sprow LANMANFS

*EXEC 601, 633, 637, **784**
Close *see* OSBYTE 0x77
File handle 698, 978, 983
*FORM40 828
*FORM80 828
*FREE
see Sprow LANMANFS
*FX *see* OSBYTE
*FX **777**
*HELP 601
LANMANAGER
see Sprow LANMANFS
*INFO *see* Acorn DFS
*KEY 601
*LANMAN *see* Sprow
see Sprow LANMANFS
*LI. *see* *LINE
*LIB *see* Acorn DFS
see Sprow LANMANFS
*LINE 601, **778**
*LIST *see* Acorn DFS
*LOAD 457, 601, **784**
*M. *see* *MOTOR
*MODE
see Lancaster Assembler
*MOTOR 601, 647, **778**
*MOUNT
see Sprow LANMANFS
*OPT 601, 649, **785**
*PING
see Sprow LANMANFS
*REMOTE 700
*RENAME *see* Acorn DFS
see Sprow LANMANFS
*ROM 601, 650
Tape or ROM switch 684, 978, 983
*RUN **786**
*SAVE 458, 601, **786**
*SDISC *see* Sprow *SDISC
*SPOOL 601, 633, **786**
Close *see* OSBYTE 0x77
File handle 698, 978, 983
*STATUS
DNS *see* Sprow *STATUS
EMADDR
see Sprow *STATUS
EMADVERTISE
see Sprow *STATUS
EMLINK *see* Sprow *STATUS
EMMASK
see Sprow *STATUS
GATEWAY
see Sprow *STATUS
*T. *see* *TAPE
*TAPE 601, 650, **778**
Tape or ROM switch 684, 978, 983
*TV 601, **779**
*WIPE
see Sprow LANMANFS
+3DOS 853, **1069**
16-Bit **1069**
1977 Trinity **1069**
1MHz Bus
Select main/cartridge
see OSBYTE 0x6B
2 key rollover 587
3” floppy **1069**
3.5” floppy **1070**
32-Bit **1069**
3V3 **1070**
5.25” floppy **1070**
see Media
5V **1070**
64-Bit **1070**
6502 second processor
see Second Processor
65C02 80
680X0 **1073**

6845 CRTC **1073**
68K **1074**
8" floppy **1074**
8-Bit **1074**

Numbers

6309 **1070**
6501 **1070**
6502 65, 69, 94, **1071**
  6502A **1071**
  6502AC **1071**
  6502C **1071**
  65C02 **1073**
  65C102 **1073**
  6503 **1071**
  6504 **1071**
  6505 **1071**
  6506 **1072**
  6507 **1072**
  6508 **1072**
  6509 **1072**
  6510 **1072**
  6512 **1072**
    6513 **1072**
    6514 **1072**
    6515 **1072**
  65802 **1072**
  65816 **1073**
  JMP bug **171**
6510 1017
6809 **1073**
8080 **1074**
  8080A **1074**
  8085 **1074**
8086 **1074**
  80186 **1074**
8500 **1074**
32016 **1069**
65816 84
68000 **1073**

A

Acorn **1074**
Acorn ADFS **811**, **1075**
Acorn Atom **1075**
Acorn Atom DOS **1075**
Acorn Business Computer **1075**
Acorn Communicator **1075**
Acorn DFS **791**, **1075**
  *ACCESS **791**
  *BACKUP **792**
  *BUILD **792**
  *CAT **793**
    see *CAT
  *COMPACT **793**
  *COPY **794**
  *DELETE **794**
  *DESTROY **795**
  *DIR **795**
  *DRIVE **795**
  *DUMP **796**
  *ENABLE **796**
  *EXEC *see* *EXEC
  *INFO **796**
  *LIB **797**
  *LIST **797**
  *LOAD *see* *LOAD
  *OPT 1 798
  *OPT 4 798
  *RENAME **799**
  *RUN *see* *RUN
  *SAVE *see* *SAVE
  *SPOOL *see* *SPOOL
  boot type 798, 803
  catalog 800
  lock file **791**, 805
  sector 800
Acorn Electron 3, 149, **1075**

Acorn NFS **825**
Acorn Proton **1076**
Acorn System **1076**
Active low **1076**
ADC **1076**
  Conversion type 690, 978, 980
  Current channel 689, 978, 981
  Enable/Disable
    see OSBYTE `0xA3`
  Force conversion
    see OSBYTE `0x11`
  Max ADC channels 689, 978, 982
  Read channel
    see OSBYTE `0x80`
  Read port *see* OSBYTE `0x80`
  Select channel
    see OSBYTE `0x10`
Address
  D800 839
Address Bus **1076**
ADFS *see* Acorn ADFS
ADVAL 639
Advanced Disk File System
  see Acorn ADFS
AIM **1076**
All-in-one **1076**
Amiga **1077**
Amiga A1000 **1077**
Amiga A1200 **1077**
Amiga A500 **1077**
Amiga A600 **1077**
AmigaOS **1077**
AmigaOS 4 **1077**
Amstrad **1078**
Analogue 1001
Angus Duggan
  see Duggan HDFS
Apple **1078**
Apple 1 **1078**
Apple II **1078**
Apple IIgs **1079**
Apple III **1079**
Arcade **1079**
Archimedes **1079**
Arduino **1079**
ARM **1079**
ARPANET **1080**
Arthur **1080**
ASCII **1080**
Assembler **1080**
Assemblers
  ADE Assembler 453
  BBC Basic Assembler 397
  BeebASM 409
  Lancaster Assembler **447**
Assembly Language **1080**
Atari **1080**
Atari SALLY **1080**
Atari ST **1081**
Atom DOS *see* System DFS
AY-3-8912 **1081**

B

BASIC **3**, **1081**
Battery Backed RAM **1081**
Baud **1081**
BBC Basic 3, **1081**
  [ ] 397, **405**
  abs 20
  Acorn Electron 13
  acs 20
  ADVAL 639
    (0) 638
    (n) 639
  adval 21
  Assembler
    see BBC Basic Assembler
  auto 15
  BBC Microcomputer 13

delete 16
else *see* if
if 19
INKEY
-256 643
key code 642
timeout 641
renumber 17
then *see* if
BBC Basic Assembler
[ ] 397, **405**
ASL 400
EQUB **405**
EQUD **406**
EQUS **406**
EQUW **405**
JMP 399
O% **401**
OPT 403, **406**
P% **401**
BBC Master 128 **1082**
BBC Master 512 **1082**
BBC Master Compact **1082**
BBC Master Econet Terminal **1082**
BBC Master Scientific **1082**
BBC Master Turbo **1082**
BBC Microcomputer 3, 81, 149, **1082**
BBC Model A **1082**
BBC Model B **1083**
BBC Model B+ **1083**
BCD **70**, **1083**
BeebASM Assembler
Command line
-D **411**
-boot **411**
-d **411**
-di **411**
-do **411**
-i **411**
-o **411**
-opt **411**
-title **411**
-v **411**
-vc **411**
-w **411**
Commodore C64 409
Constants
CPU **438**
FALSE **438**
P% **438**
PI **438**
TRUE **438**
Directives
ALIGN **421**
ASM **421**
ASSERT **421**
CLEAR **422**
COPYBLOCK **422**
CPU **422**
EQUB **423**
EQUD **423**
EQUS **423**
EQUW **423**
ERROR **424**
FOR **424**
GUARD **426**
IF **427**
INCBIN **429**
INCLUDE **429**
MACRO **429**
MAPCHAR **432**
ORG **433**
PRINT **433**
PUTBASIC **434**
PUTFILE **434**
PUTTEXT **435**
RANDOMIZE **435**
SAVE **436**
SKIP **437**
SKIPTO **437**

SOURCELINE 437
Functions
<val 418
>val 418
ABS 439
ACS 439
ASC 439
ASN 439
ATN 439
CHR$ 440
COS 440
DEG 440
EVAL 440
EXP 440
HI(val) *see* >val
INT 441
LEFT$ 441
LEN 441
LN 441
LO(val) *see* <val
LOG 442
LOWER$ 441
MID$ 442
NOT 442
RAD 442
RIGHT$ 443
RND 443
SGN 443
SIN 443
SQR 444
STR$ 444
STR$~ 444
STRING$ 444
TAN 445
TIME$ 445
UPPER$ 445
VAL 445
Labels
{} *see* =413
.*label 414
.^label 415
Local 413
Operators 417
AND 417
« 417
» 417
DIV 417
EOR 417
MOD 417
OR 417
Variables
CALLSTACK$ 433
FILELINE$ 433
BeOS 1083
Big-Endian 69, 1083
Binary 66, 1083
Binary Coded Decimal
see BCD
Bit 1083
Bluetooth™ 1083
Brandy Basic 1083
BSD
License 1084
Operating System 1084
BT 1084
BT Merlin Tonto 1084
Buffer
Examine status
see OSBYTE 0x98
Flush *see* OSBYTE 0x15
Flush all *see* OSBYTE 0x0F
Insert into *see* OSBYTE 0x8A
Insert into input
see OSBYTE 0x99
Read from *see* OSBYTE 0x91
Read status
see OSBYTE 0x80
Buffer Overrun 769
Byte 68, 1084
double *see* word

long 1097
nibble *see* Nibble
word **69**, 1120

C

C 98, 100
C128 **1084**
C64 81, **1085**
Edge Connector 1016
Cambridge Co-Processor **1085**
Cambridge Workstation **1085**
Carry Flag **1085**
Cartridge **1085**
Acorn Electron 1003
BBC Master 1004
Master Compact 1005
Cassette 1000, **1085**
Cassette deck **1085**
Cassette File System **787**
CBM PET **1085**
CBM-II **1085**
CD Drive **1086**
Centisecond **1086**
Centronics **1086**
CFS *see* Cassette File System
CGA **1086**
Character
Explosion state 683, 978, 981
Characters
Explode definitions
see OSBYTE `0x14`
Reset definitions
see OSBYTE `0x14`
see OSBYTE `0x19`
CIFS
see Common Internet Filing System
CISC **1086**
Clock **1086**
CMOS **1087**
Read *see* OSBYTE `0xA1`
Write *see* OSBYTE `0xA2`
Colour
Flash mark duration
see OSBYTE `0x09`
Flash space duration
see OSBYTE `0x0A`
Reset Flash Cycle
see OSBYTE `0x9A`
Commodore **1087**
64 837
PET 837
Commodore 64 3
Commodore Basic 3, **1087**
Common Internet Filing System
see Sprow LANMANFS
Composite Video **1087**
CP/M 373, **1088**
CP/NOS **1088**
CPC464 **1088**
CPC6128 **1088**
CPC664 **1088**
CPU **1089**
CRT **1089**
CTS **1089**
Cursor
Enable/disable editing
see OSBYTE `0x04`
Get character at text cursor
see OSBYTE `0x87`
Get Text Position
see OSBYTE `0x86`

D

Data Bus **1089**
Decimal **65**, **1089**
DFS *see* Acorn DFS
Digital **1089**

Digital Research **1089**
Dip **1090**
Double **1090**
double byte *see* Word
DRAM **1090**
Driver **1090**
Dsub **1090**
Duggan HDFS **827**, **1090**

E

Econet 825, 1002, **1090**
  alternatives 813
  OS intercept 706, 979, 981
  Read char intercept 707, 979, 981
  Write char intercept 707, 979, 981
EEPROM **1090**
  Read *see* OSBYTE `0xA1`
  Write *see* OSBYTE `0xA2`
EGA **1091**
Endian
  Big *see* Big-Endian
  Little *see* Little-Endian
Endianess **69**, **1091**
Enter language ROM
  see OSBYTE `0x8E`
EPROM **1091**
Escape
  Clear condition
    see OSBYTE `0x7C`
    see OSBYTE `0x7E`
  Set condition
    see OSBYTE `0x7D`
ESP32 **1091**
Eurocard **1091**
Event
  Disable *see* OSBYTE `0x0D`
  Enable *see* OSBYTE `0x0E`
  Vertical Sync
    see OSBYTE `0x13`
Explode character definition RAM
  see OSBYTE `0x14`

F

File
  Check for EOF
    see OSBYTE `0x7F`
File handle **1091**
File System **1091**
Filesystem
  Get high order address
    see OSBYTE `0x82`
  Make temp permanent
    see OSBYTE `0x6D`
  Set options *see* *OPT
Firm key **1092**
Firmware **1092**
Floppy controller **1092**
Floppy disk **1092**
Floppy drive **1092**
FRED
  Read from *see* OSBYTE `0x92`
  Write to *see* OSBYTE `0x93`
FreeBSD **1092**
Function Key
  Reset Soft Keys
    see OSBYTE `0x12`
Function Keys
  Cursor keys
    see OSBYTE `0x04`

G

GEM **1092**
Genlock **1092**
Get Machine type
  see OSBYTE `0x00`

see OSBYTE 0x81
Get OS Version
see OSBYTE 0x00
see OSBYTE 0x81
GitHub 410
\glsentrynamebbcbasic
ENVELOPE 761
SOUND 759
TIME 755
GNU **1093**
Go 98, 100
GoTek **1093**
GPIO **1093**

H

Handshake **1093**
Hard drive **1093**
HDFS *see* Duggan HDFS
HDMI **1093**
Hexadecimal **66**, **1093**
High **1094**
HIMEM 645
see OSBYTE 0x84
For specific mode
see OSBYTE 0x85
Hitachi **1094**
HMOS **1094**

I

I cannot run this code 672
IBM **1094**
IBM PC **1094**
IBM PC-Jr **1094**
IDC **1094**
IDE **1095**
IEEE **1095**
File system **834**
INKEY
Acorn Electron **592**
Intel **1095**

J

Jiffy **1095**
JIM
Read from *see* OSBYTE 0x94
Write to *see* OSBYTE 0x95

K

Kempston **1095**
Kernal **837**, **1095**
Kernel **1096**
Keyboard
Auto-repeat delay 697, 978, 981
Auto-repeat rate 697, 978, 981
Char interpretation 716–718, 979, 981
Cursor editing 731, 980
Disable 700, 978, 981
ESCAPE action 724, 979, 981
ESCape character 715, 979, 981
ESCAPE effects 725, 979, 981
Func key status 718–723, 979, 981
Keypad base 732, 980, 982
Keys pressed
see OSBYTE 0x78
Read with time limit
see OSBYTE 0x81
repeat
delay *see* OSBYTE 0x0B
rate *see* OSBYTE 0x0C
Scan for key
see OSBYTE 0x79
see OSBYTE 0x81

Semaphore 678, 978, 981
Soft key pointer 729, **729**, 980, 982
Status flags 701, 978, 981
TAB character 714
Update LEDs
see OSBYTE 0x76
Keypad 1025

L

Lancaster Assembler **447**, **1096**
*ASSEMBLE **452**
*BASIC 449
*CLI **449**
*CONFIGURE
LANG 449
*MODE **449**
*VIEW 449
*WORDWISE 449
$CLST **525**
$DEFCLST **525**
$DEFLST **525**
$FLEVEL **525**
$FS **525**
$LINE **525**
$LST **525**
$MC **525**
$MLEVEL **525**
$OBJECT **525**
$OS **525**
$OSHWM **525**
$PLENGTH **525**
$PRINTING **526**
$PWIDTH **525**
$SOURCE **526**
$TIME **526**
$TTL **526**
$VERSION **526**
ADE Assembler 453
AGO 509
AIF 509
ALEN 511, **513**
ASC **483**, 485
ASET 511, **512**, 513
ASTR 511, **514**
ASTRIP 511, **514**
CASC **484**
CHN 493
CHR 475
CLST 472
CODE **486**
CSTR **484**
DB **480**
DDB **480**
DEFPARS **522**
DEND 489
DFB **480**, 482, 485
DFDB **480**, **485**
DFW **480**
Directives
AGO 529
AIF 529
AIFDEF 529
AIFNDEF 530
ALEN 530
ASC 530
ASET 530
ASTR 531
ASTRIP 531
CASC 531
CHN 532
CHR 532
CLST 533
CODE 533
CSTR 534
DB 534
DDB 534
DEFPARS 535
DEND 535

DFB 535
DFDB 536
DFS 536
DFW 536
DISP 536
DISP1 536
DISP2 537
DO 537
DS 537
DSECT 538
DW 538
ELSE 539
ENDM 539
EQU 539
EXEC 540
FI 540
FIN 540
HEX 541
IF 541
IFDEF 541
IFNDEF 542
INCLUDE 542
INFO 542
LFCOND 543
LOAD 543
LST 544
MACRO 545
MEXIT 545
MLEN 545
MSET 546
MSTR 546
MSTRIP 546
MSW 546
ORG 547
PAGE 547
QUERY 548
REP 548
SFCOND 548
SKP 549
STOP 549
STR 549
SYSCLI 550
SYSVDU 551
SYSVDU1 551
SYSVDU2 551
TABS 552
TIME 553
TTL 552
WAIT 553
WAIT1 553
WAIT2 553
DISP 478, 499
DISP1 499
DISP2 499
DO 496
ELSE 496
FIN 496
DS 490
DSECT 489
DW **480**
ELSE 495
EQU 489
FI 495
FIN 496
HEX **485**
IF 495
ELSE 495
FI 495
IFDEF 497
IFNDEF 497
INCLUDE 494
INFO 499
LFCOND 497
LST 469
MACRO 503
ENDM 503
MEXIT **523**
MLEN 511, **513**
MSET 511, **512**, 513
MSTR 511, **514**

MSTRIP 511, **514**
MSW 492
ORG **487**
PAGE 473
QUERY 500
REP 475
SFCOND 497
SKP 475
STOP 502
STR **483**
SYSVDU 477
SYSVDU1 477
SYSVDU2 477
TABS 471
TIME 474
TTL 474
View 451
WAIT 500
WAIT1 500
WAIT2 500
WordWise 451
Language
Enter ROM
see OSBYTE `0x8E`
LANMANFS *see* SPROW
Laptop **1096**
Latch **1096**
LIFO **1096**
Light Gun 1025
Linux 67, 409, **1096**
Little-Endian **69**, **1097**
Long **1097**
Low **1097**

M

Machine Code **1097**
Macintosh **1097**
MacOS 67, 409, **1097**
Mainframe **1098**
Maskable interrupt **1098**
Master
LK12 1010
LK21 1010
LK6 1009
Master Turbo-Card
see Second Processor
Matrix Brandy Basic **1098**
MCGA **1098**
MDA **1098**
Media
5.25” floppy **780**
Memory
Size *see* OSBYTE `0x9D`
Memory switching 849, 851
Microcomputer **1098**
Microcontroller **1099**
Microdrive **1099**
Microprocessor **1099**
Microsoft **1099**
MIDI 1024, **1099**
Minicomputer **1099**
MiSTer **1099**
Modern Computing **1099**
MOS **1100**
Motherboard **1100**
Motorola **1100**
MS-Dos **1100**
MS-Windows **1100**
MSBasic 3, **1100**
Musical Frequencies **58**

N

National Semiconductor **1100**
Negation **73**
Negative Flag **1100**
NetBSD **1101**
NFS *see* Acorn NFS
Nibble **68**, **1101**

NitrOS-9 **1101**
NMI **1101**
NMOS **1101**
number
BCD **70**
Binary **66**
Byte **68**
Decimal **65**
Hexadecimal **66**
Negation **73**
Nibble *see* Nibble
Octal **67**
One's complement **73**
Two's complement **71**
nybble *see* Nibble

O

Octal **67**, **1101**
One's complement **1101**
One's complement **73**
Open Hardware **1102**
Open Source **1102**
OpenBSD **1102**
Opus DDOS **830**, **1102**
OS **1102**
OS-9 **1102**
OSASCI **597**
OSBGET **599**
OSBPUT **599**
OSBYTE **603**
`0x00` Get OS Version **608**
`0x01` Write user flag **609**
`0x02` Specify input stream **610**
`0x03` Specify output stream **611**
`0x04` Enable/disable cursor editing **612**
`0x05` Select printer destination **613**
`0x06` Write printer ignore character **613**
`0x07` Set \glsentrynamers423 receive rate **614**
`0x08` Set \glsentrynamers423 transmit rate **614**
`0x09` Set duration of first colour **615**
`0x0A` Write duration of second colour **615**
`0x0B` Set keyboard auto-repeat delay **616**
`0x0C` Set keyboard auto-repeat rate **616**
`0x0D` Disable event **617**
`0x0E` Enable event **617**
`0x0F` Flush buffers **618**
`0x10` Select ADC channel **618**
`0x11` Force ADC conversion **619**
`0x12` Reset soft function keys **619**
`0x13` Wait for vertical sync **620**
`0x14` Explode character definition RAM **620**
`0x14` Reset character definitions **621**
`0x15` Flush specific buffer **622**, **622**
`0x16` Increment polling semaphore **623**
`0x17` Decrement polling semaphore **623**
`0x18` Select external Sound **624**
`0x19` Restore default font definitions **624**
`0x44` Test for sideways RAM **625**
`0x45` Sideways RAM allocation **626**
`0x6B` Select 1MHz bus/cartridge

627
0x6C Select screen memory for direct access 628
0x6D Make temporary file system permanent 628
0x70 Select memory for VDU 629
0x71 Select memory for display 630
0x72 Write shadow memory use 630
0x73 Blank or restore palette 631
0x74 Reset sound system 631
0x75 Read VDU status 632
0x76 Make LEDs reflect keyboard status 633
0x77 Close *SPOOL/*EXEC files 633
0x78 Write current keys pressed 634
0x79 Perform keyboard scan 634
0x7A Perform keyboard scan from 0x10 635
0x7B Printer driver going dormant 636
0x7C Clear ESCAPE condition 636
0x7D Set ESCAPE condition 637
0x7E Acknowledge ESCAPE, clear + effects 637
0x7F Check for EOF 638
0x80 Read ADC channel/buffer status 638, 640
0x81 Read key with time limit 641
0x81 Read machine type 643
0x81 Scan for specific key 642
0x82 Read machine high order address 644
0x83 Read top of OS RAM 644
0x84 Read top of user RAM 645
0x85 Read HIMEM for given mode 645
0x86 Read input cursor pos 646
0x87 Read char at text cursor and screen mode 646
0x88 *CODE 647
0x89 *MOTOR 647
0x8A Insert into buffer 648, 648
0x8B *OPT 649
0x8C *TAPE 650
0x8D *ROM 650
0x8E Enter language ROM 651
0x8F Issue paged ROM service request 651
0x90 *TV 779
0x90 *TV 652
0x91 Get character from buffer 653, 653
0x92 Read from FRED 654
0x93 Write to FRED 655
0x94 Read from JIM 655
0x95 Write to JIM 655
0x96 Read from SHIELA 656
0x97 Write to SHIELA 656
0x98 Examine buffer status 657
0x99 Insert character into input buffer 660
0x9A Reset flash cycle 661
0x9A Write to video ULA 661
0x9B No operation 662
0x9B Write to video ULA palette register 662
0x9C Read/Write 6850 control register 663
0x9D Fast Tube BPUT 665
0x9E Read from speech processor 665

0x9E Reserved 666
0x9F Reserved 667
0x9F Write to speech processor 666
9F Write to speech processor@0x9F Write to speech processor 667
0xA0 Read VDU variable 668, 668
0xA1 Query \glsentrynameeeprom size 670
0xA1 Read CMOS \glsentrynameram 669
0xA1 Read \glsentrynameeeprom 669
0xA2 Write CMOS \glsentrynameram/\glsentrynameeeprom 670
0xA3 Enable or Disable printer and ADC 671, 671
0xA3 Reserved for application software 671
0xA4 Check for 6502 code 672
0xA5 Read output cursor position 673
0xA6 Base of MOS variables 673
0xA8 ROM Pointer table 674
0xAA ROM Information table 675
0xAC Keyboard Translation Table 676
0xAE VDU Variables Table 676
0xB0 CFS timeout counter 677
0xB1 Input source 677
0xB2 Keyboard semaphore 678
0xB3 Default OSHWM 679
0xB3 Paged ROM polling semaphore 680
0xB3 Read Write primary OSHWM 710
0xB4 Current OSHWM 681
0xB5 RS-423 mode 682
0xB6 Character Explosion State 683
0xB6 Printer Ignore Character 683
0xB7 TAPE or ROM Switch 684
0xB8 Video ULA control register 684
0xB9 Paged ROM polling semaphore 686
0xB9 Video ULA palette register 685
0xBA ROM active on last BRK 687
0xBB BASIC ROM number 688
0xBC Current ADC channel being sampled 689
0xBD Maximum ADC channel number 689
0xBE ADC conversion type 690
0xBF RS-423 Use Flag 693
0xC0 RS-423 6840 control register 694
0xC1 Read Flash Counter 695
0xC2 Flash Mark Colour Duration 696
0xC3 Flash Space Colour Duration 696
0xC4 Keyboard Auto-repeat delay 697
0xC5 Keyboard Auto-repeat rate 697
0xC6 *EXEC File Handle 698
0xC7 *SPOOL File Handle 698
0xC8 Escape and Break Effect 699
0xC9 Keyboard Disable 700

0xCA Keyboard Status 701
0xCB Electron ULA IRQ mask 703
0xCB RS-423 Handshake Level 702
0xCC Firm Key Pointer 704
0xCC RS-423 Suppression Flag 703
0xCD Firm Key string length 706
0xCD RS-423 Selection Flag 705
0xCE Econet OS call interception status 706
0xCF Econet read char interception status 707
0xD0 Econet write char interception status 707
0xD1 Speech suppression status 708
0xD2 Sound suppression status 708
0xD3 Sound BELL channel 709
0xD4 Sound BELL amplitude/envelope 709
0xD5 Sound BELL pitch 710
0xD6 Sound BELL duration 710
0xD7 Startup message and Boot Options 711
0xD8 Soft Key Length 712
0xD9 Lines printed since last page pause 712
0xDA Number items in VDU queue 713
0xDB External sound flag 715
0xDB TAB character 714
0xDC ESCape character 715
0xDD Input buffer interp Cx 716
0xDE Input buffer interp Dx 717
0xDF Input buffer interp Ex 717
0xE0 Input buffer interp Fx 718
0xE1 Function Key status 718
0xE2 Function Key + A\dotsP status 720
0xE2 Function Key + SHIFT status 719
0xE3 Function Key + CTRL status 721
0xE3 Function Key + Q\dotsZ status 722
0xE4 Function Key + CTRL SHIFT status 723
0xE5 Escape action flag 724
0xE6 Escape effects flag 725
0xE7 Reserved for hardware expansion 726
0xE7 User VIA IRQ bit mask 726
0xE8 RS-423 6850 IRQ bit mask 727
0xE8 Sound semaphore 727
0xE9 Soft key pointer 729
0xE9 System VIA IRQ bit mask 728
0xEA Tube present flag 730
0xEB Speech System present flag 730
0xEC Output stream destination 731
0xED Cursor editing status 731
0xEE Numeric Keypad base 732
0xEF Shadow memory use 733
0xF0 Country Code 734
0xF1 User Flag 735
0xF2 Last value written to ULA 737
0xF2 Serial ULA control register 736
0xF3 Time Clock Switch 738
0xF4 Soft key consistency flag 739
0xF5 Printer destination 740

0xF6 Printer ignore character **740**
0xF7 Break Intercept **741**
0xFA Select memory for VDU **742**
0xFB Select memory for display **742**
0xFC Current language ROM **743**
0xFD Last reset type **744**
0xFE Keypad Shift Key Flag **746**
0xFE RAM Size **745**
0xFE Reserved **745**
0xFF Startup Options **747**, **748**
BBC Master Compact **749**
BBC Master 128 **749**
OSCLI **600**
OSEVEN **602**
OSHWM 644
see OSBYTE 0x83
OSNEWL **596**
OSRDCH **598**
OSRDRM *see* OSRDSC
OSRDSC **602**
OSWORD **751**
0x00 Read line from current input stream **753**
0x01 Read system clock **755**
0x02 Write system clock **755**
0x03 Read interval timer **756**
0x04 Write interval timer **756**
0x05 Read byte of I/O processor memory **757**, **758**
0x06 Write byte of I/O processor memory **758**
0x07 Perform a SOUND command **759**
0x08 Define an ENVELOPE **761**
0x09 Read pixel value **764**
0x0A Read character definition **765**
0x0B Read palette for given logical colour **766**
0x0C Write palette for given logical colour **766**
0x0D Read previous and current graphics cursor position **767**
0x0E Read real time clock **768**, **769**, **770**, **770**
0x0F Write real time clock **772**, **772**
DFS 0.90 bug **752**
0xFF Read/Write I/O processor memory Z80 **773**
Unknown calls **752**
OSWRCH **596**

P

Page **1102**
Paged ROM **1103**
Active last BRK 687, 978, 982
BASIC ROM number 688, 978, 980
Language *see* OSBYTE 0x8E
Polling semaphore 623, 680, 686
Service Request *see* OSBYTE 0x8F
Paged Rom
Service Call
0x03 634
0x10 633
0x15 623, 680, 686
Palette
Blank or Restore *see* OSBYTE 0x73
Panos **1103**
Parallel **1103**
PATA **1103**

PC **1103**, **1105**
PC-Dos **1103**
PCMIA **1103**
PDA **1103**
PDP-11 67, **1104**
Peripheral **1104**
PETSCII **1104**
PITube **1104**
Plus 1 **1104**
Plus 3 **1104**
Polling Semaphore
  Decrement *see* OSBYTE `0x17`
  Increment *see* OSBYTE `0x16`
  Varaible *see* OSBYTE `0x43`
    see OSBYTE `0x49`
Ports
  Analogue 1001
  Cartridge
    Acorn Electron 1003
    BBC Master 1004
    Master Compact 1005
  Cassette 1000
  Commodore
    RS-232 1015
    Serial Bus 1013
    User Port 1014
  Econet 1002
  Edge Connector
    C64 1016
    Spectrum 1026
    ZX81 1019
  MIDI
    Sinclair 128K 1024
  RGB
    BBC Micro 999
    Sinclair 128K 1023
  RS-232
    Sinclair 128K 1024
  RS-423 999
  Sinclair 128K
    Keypad 1025
    Light Gun 1025
    MIDI 1024
    RS-232 1024
POSIX **1105**
PowerPC **1105**
Printer **1105**
  Destination
    see OSBYTE `0x05`
  Driver dormant
    see OSBYTE `0x7B`
  Enable/Disable
    see OSBYTE `0xA3`
  Ignore char
    see OSBYTE `0x06`
  Ignore character 683, 978, 982
Prism VTX5000 **1105**
PROM **1105**
Pyromaniac **1105**

R

RAM **1105**
  Size *see* OSBYTE `0x9D`
RAM Disk 849, 853, **1105**
Raspberry PI **1106**
Raspberry PI Pico **1106**
recursion 85
Register **1106**
Research Machines **1106**
Restore default font definitions
  see OSBYTE `0x19`
Retro Computing **1106**
Reuters **1106**
Reuters APM **1106**
Reuters OS **1106**
RF **1106**
RGB 999, 1023, **1107**
RGBI **1107**
Ribbon **1107**

RISC **1108**
RISC-OS **1108**
RM 380 **1108**
RM 380Z **1108**
RM 480Z **1108**
RM Nimbus **1108**
Rollover **1108**
ROM **1109**
ROM Filesystem **789**
ROM FS **1109**
Rotate **1109**
RS-232 1015, 1024, **1109**
  RS-232C **1109**
RS-423 999, **1109**
  6840 control register 694, 978, 982
  Handshake level 702, 979, 982
  IRQ bit mask 727, 979, 982
  Keyboard mode 682, 978, 982
  Read/Write 6550 control register
    see OSBYTE 0x9C
  Receive rate
    see OSBYTE 0x07
  Selection flag 705
  Suppression flag 703
  Transmit rate
    see OSBYTE 0x08
  use flag 693, 978, 982
RTS **1110**
RunCPM **1110**

S

SAS **1110**
SATA **1110**
Scanner **1110**
Screen
  Force shadow memory
    see OSBYTE 0x72
  Get current mode
    see OSBYTE 0x87
  Select for access
    see OSBYTE 0x6C
  Select for Display 742, 980, 982
  Select for display
    see OSBYTE 0x71
  Select for VDU 742, 980, 982
    see OSBYTE 0x70
  Shadow memory use 733, 980, 982
  Write to Video ULA
    see OSBYTE 0x9A
  Write to Video ULA palette
    see OSBYTE 0x9B
SCSI **1110**
SDL **1110**
Second Processor 784, 786, **1111**
  6502 second processor 457
  Master Turbo-Card 457
Sega **1111**
Sega Master System **1111**
Select external Sound
  see OSBYTE 0x18
Serial **1111**
Serial Bus 1013, **1111**
Service ROM **1112**
SHIELA
  Read from *see* OSBYTE 0x96
  Write to *see* OSBYTE 0x97
Shift **1112**
Sideways RAM **1112**
  Addressing mode
    see OSBYTE 0x45
  Test for *see* OSBYTE 0x44
Sinclair **1112**
Sinclair 128K
  MIDI 1024
  RS-232 1024
Sinclair Basic **1112**
  data 26

for **27**
go sub 29
go to 30
if 28
let 31
next *see* for
read 32
restore 33
return 34
step *see* for
then *see* if
Sinclair QL **1112**
Single Board Computer **1112**
Solidisk DFS **1113**
Sound
Bell amplitude 709
Bell channel 709
Bell duration 710
Bell pitch 710
External flag 715, 979, 981
Reset system
see OSBYTE `0x74`
Select alternate system
see OSBYTE `0x18`
Semaphore 727, **728**, 979, 982
Suppression status 708, 979, 982
Spectrum 3, 372, **1113**
see also Spectrum 48K
see also Spectrum+
OS Variable
0x5B5C 849, 851
0x5B67 851
Port
1FFD **851**, 852
7FFD **849**, 850, **851**, 852
RAM Disk 849
Spectrum +2 **1113**
Memory switching **849**
Spectrum +2A **1113**
Memory switching **851**
Spectrum +3 **1113**
see also +3DOS
Memory switching **851**
Spectrum 128K 3, 81, **1113**
Edge Connector 1026
Memory switching **849**
Spectrum 48K
Edge Connector 1026
Spectrum+ **1113**
Speech
Present flag 730, 980, 983
Read *see* OSBYTE `0x9E`
suppression 708, 979, 983
Write *see* OSBYTE `0x9F`
Spreadsheet **1114**
Springboard **1114**
SPROW
*ACCESS **820**
*BYE *see* *DISMOUNT
*CDIR **820**
*CONFIGURE DNS **816**
*CONFIGURE EMADDR **816**
*CONFIGURE EMADVERTISE **815**
*CONFIGURE EMLINK **814**
*CONFIGURE GATEWAY **816**
*COPY **820**
*DESTROY **821**
*DIR **821**
*DISMOUNT **821**
*DRIVE **821**
*EMINFO **817**
*EMTEST **817**
*ENABLE **822**
*FREE **822**
*HELP LANMANAGER **822**
*LANMAN **822**
*LIB **823**
*MOUNT **823**
*PING **818**

*RENAME **823**
*SDISC *see* *MOUNT
*STATUS
DNS **819**
EMADDR **818**
EMADVERTISE **818**
EMLINK **818**
EMMASK **819**
GATEWAY **819**
*WIPE **824**
Ethernet 813, 1002
Ethernet cable 1002
LANMANFS **813**
Stack **1114**
Static RAM **1114**
STM32 **1114**
Stream
Select input
see OSBYTE `0x02`
Select output
see OSBYTE `0x03`
Subprogram **1114**
System DFS **827**, **1114**

T

Tandy **1115**
Tape deck **1115**
Tape Filesystem
see Cassette File System
TCP/IP 70, **1115**
Teensy **1115**
Telesoft **834**
Teletext
Telesoft **834**
Text Cursor
Get character at cursor
see OSBYTE `0x87`
Get Position
see OSBYTE `0x86`
This is not a language 672
TI **1115**
TI Basic 3, **1115**
CALL 51
CHAR **51**
COLOR **52**
GCHAR **53**
HCHAR **55**
SCREEN **55**
SOUND **56**
VCHAR **59**
CALL CLEAR **51**
DATA 38
DISPLAY 38
ELSE *see* IF
END 39
FOR **40**
GO TO 41
GOTO 41
IF 42
INPUT 44
LET 42
Musical Frequencies **58**
NEXT 43
see FOR
ON GO TO 43
ON GOTO 43
PRINT 45
READ 48
REM 48
RESTORE 48
STEP *see* FOR
STOP 49
Subprogram *see* CALL
THEN *see* IF
TI Calculator **1115**
TI Speech Synthesizer **1115**
TI-99/4A 3
TI99/4A **1115**
TI99/4 **1115**

Torch Computers **1116**
TRS-80 **1116**
TRS-80 Color Computer **1116**
TTL **1116**
Tube 784, 786, **1116**
  Present flag 730, 980, 983
TV **1116**
Two's complement **1116**
Two's complement **71**

U

U.V. **1118**
UHF **1117**
ULA **1117**
Unix 67, **1117**
Unix-Time **1117**
USB 841, **1117**
User Flag *see* OSBYTE `0x01`
User land **1118**
User Port 1014
USERV
  A=0 *CODE 647, **777**
  A=1 *LINE **778**

V

V-Sync **1119**
VDU **1118**
  19 **992**
  20 **992**
  23 992
    Set cursor state 992
  Read status
    see OSBYTE `0x75`
  read variable
    see OSBYTE `0xA0`
  Select screen for access
    see OSBYTE `0x70`
  Variables table 676, 978, 983
Vertical Sync
  Wait for *see* OSBYTE `0x13`
VGA **1118**
VHF **1118**
VIC-II 1017
Vic20 **1118**
View **1119**
Viewdata **1119**
ViewSheet **1119**

W

Wait for vertical sync
  see OSBYTE `0x13`
Watford 62 DFS **833**
Watford 62DFS **1119**
Watford DDFS **833**, **1119**
Watford DFS **831**, **1119**
Wi-Fi™ **1120**
WIMP **1120**
Windows 409
Word **69**, **1120**
Word Processor **1120**

X

Xenix **1120**
XWindows **1120**

Y

Y2K **1120**
Y2K38 **1121**

Z

Z80 65, 69, **1121**

LD r,(HL) **321**
Z80-OS **1121**
Zero Flag **1121**
Zero-page **1121**
Zilog **1122**
ZX-80 **1122**
ZX-81 3, **1122**
Edge Connector 1019
ZX-Printer **1122**
ZX Interface 1 **1122**

Notes

Notes

Notes

Notes